A Performance Analysis of the Globus Toolkit's Job Submission, GRAM

Ioan Raicu[*] Catalin Dumitrescu[*] Ian Foster[+*]

[*]Computer Science Department
The University of Chicago
{iraicu,cldumitr}@cs.uchicago.edu

[+]Mathematics and Computer Science Division
Argonne National Laboratory
foster@cs.uchicago.edu

1 Abstract

The Globus Toolkit® (GT) is the "de facto standard" for grid computing. Measuring the performance of various components of the GT in a wide area network (WAN) as well as a local area network (LAN) is essential in understanding the performance that is to be expected from the GT in a realistic deployment in a distributed and heterogeneous environment where there might be complex interactions between network connectivity and service performance. The focus of this paper is the performance of job submission mechanism (GRAM), an essential GT component that is very likely to be used in a mixed LAN and WAN environment. We specifically tested the scalability, performance, and fairness of the pre-WS GRAM and WS-GRAM bundled with GT 3.9.4. To drive our empirical evaluation of GRAM, we used DiPerF, a DIstributed PERformance testing Framework, whose design was aimed at simplifying and automating service performance evaluation. …

2 Introduction

The Globus Toolkit is the "de facto standard" for grid computing as it has been called by numerous agencies and world recognized news sources such as the New York Times. Measuring the performance of the Globus Toolkit is important to ensure that the Grid will continue to grow and scale as the user base and infrastructure expands. Furthermore, the testing of the toolkit and grid services is difficult due to the distributed and heterogeneous environment Grids are usually found in. Performing distributed measurements is not a trivial task, due to difficulties 1) *accuracy* – synchronizing the time across an entire system that might have large communication latencies, 2) *flexibility* – in heterogeneity normally found in WAN environments and the need to access large number of resources, 3) *scalability* – the coordination of large amounts of resources, and 4) *performance* – the need to process a large number of transactions per second. In attempting to address these four issues, we developed DiPerF, a DIstributed PERformance testing Framework, aimed at simplifying and automating service performance evaluation. DiPerF coordinates a distributed pool of machines that run clients of a target service, collects and aggregates performance metrics, and generates performance statistics. The data collected provides information on a particular service's maximum throughput, on service 'fairness' when multiple clients access the service concurrently, and on the impact of network latency on service performance from both client and service viewpoint. Using this data, it may be possible to build empirical performance estimators that link observed service performance (throughput, response time) to offered load. These estimates can be then used as input by a resource scheduler to increase resource utilization while maintaining desired quality of service levels. All steps involved in this process are automated, including dynamic deployment of a service and its clients, testing, data collection, and the data analysis.

DiPerF is a modularized tool, with various components written in C/C++/perl; it has been tested over PlanetLab, Grid3, and the Computer Science Cluster at the University of Chicago. We have implemented several variations (ssh, TCP, UDP) of the communication between components in order to give DiPerF the most flexibility and scalability in a wide range of scenarios. We have investigated both the performance and scalability of DiPerF and found that DiPerF is able to handle up to 10,000+ clients and 100,000+ transactions per second; we also performed a validation study of DiPerF to ensure that the aggregate client view matched the tested service view and found that the two views usually matched within a few percent.

In profiling the performance of the Globus Toolkit, we investigated the performance of three main components: 1) job submission, 2) information services, and 3) a file transfer protocol. All these components of the Globus Toolkit are vital to the functionality, performance, and scalability of the grid. We specifically tested: 1) pre WS GRAM and WS GRAM included [1, 2, 3, 4, 5] with Globus Toolkit 3.2 and 3.9.4, 2) the scalability and performance of the WS-MDS Index bundled with Globus Toolkit 3.9.5, and 3) the scalability and fairness of the GridFTP server included with the Globus Toolkit 3.9.5. We also investigated the performance of two grid services: 1) DI-GRUBER, a distributed usage SLA-based broker, a grid service based on the Globus Toolkit 3.2 and 3.9.5, and 2) instance creation and message passing performance in the Globus Toolkit 3.2. Through the various test case studies we performed using DiPerF, we believe it has proved to be a valuable tool for scalability and performance evaluation studies, as well as for automated extraction of service performance characteristics.

2.1 Motivation & Goals

Multiple threads motivated me to measure the performance of the Globus Toolkit, which in turn motivated the building of DiPerF. The Globus Toolkit is the "de facto standard" for grid computing. Measuring the performance of the various components of the Globus Toolkit in a WAN as well as a LAN is essential in understanding the performance that is to be expected from the Globus Toolkit in a realistic deployment in a distributed and heterogeneous environment. Furthermore, measuring the performance of grid services in a WAN is similarly important due to the complex interactions between network connectivity and service performance.

Although performance testing is an 'everyday' task, testing harnesses are often built from scratch for a particular service. DiPerF can be used to test the scalability and performance limits of a service: that is, find the maximum offered load supported by the service while still serving requests with an acceptable quality of service. Actual service performance experienced by heterogeneous and geographically distributed clients with different levels of connectivity cannot be easily gauged based on controlled LAN-based tests. Therefore significant effort is sometimes required in deploying the testing platform itself. With a wide-area, heterogeneous deployment provided by the PlanetLab [6, 7] and Grid3 [8] testbed, DiPerF can provide accurate estimation of the service performance as experienced by such clients.

2.2 Obstacles

Automated performance evaluation and result aggregation across a distributed test-bed is complicated by multiple factors. In building DiPerF, we encountered 4 main obstacles:

- accuracy – time synchronization
- flexibility: heterogeneity in WAN environments & accessing of many resources
- scalability – coordination of many resources
- performance – processing large number of transactions per second

The accuracy of the performance metrics collected is heavily dependent on the accuracy of the timing mechanisms used and on accurate clock synchronization among the participating machines. DiPerF synchronizes the time between client nodes with a synchronization error smaller than 100ms on average. The reliability of presented results is important, especially in wide-area environments: we detect client failures during the test that could impact on reported result accuracy.

The heterogeneity normally found in WAN environments pose a challenging problem for any large scale testing due to different remote access methods, different administrative domains, different hardware architectures, and different operating systems and host environments. We have shown DiPerF to be flexible by implementing the support for three testbeds: Grid3, PlanetLab, and the University of Chicago CS cluster. Grid3 offers a testbed in which DiPerF uses the Globus Toolkit as the main method of deploying clients and retrieving performance metrics. PlanetLab offers a unique environment where there is a uniform remote access method, the client code gets deployed via rsync, and the communication is implemented via ssh, TCP, or UDP. The UChicago CS cluster is similar to that of PlanetLab, however it has the advantage of having a network file system (NFS) which make the deployment of clients almost trivial; on the other hand, the communication occurs in exactly the same manner as it does in PlanetLab.

The scalability of the framework itself is important; otherwise DiPerF will not be able to saturate a target service. We insure scalability by only loosely coupling the participating components, and by having multiple implementations of communication protocols between components. DiPerF has been designed and implemented to be scalable to 10,000+ clients that could generate 100,000+ transactions per second.

DiPerF has been measured to process up to 200,000 transactions per second via TCP and up to 15,000 transactions per second via SSH. The performance of DiPerF has been carefully tuned to use the most lightweight protocols and tools in order to achieve its goals. For example, the communication protocol built on TCP uses a single process and the select() function to multiplex the 1,000s of concurrent connections. The structures that are used to store and transfer the performance metrics have been optimized for space and efficiency. Furthermore, each TCP connection's buffering is kept to a minimum in order to lower the memory footprint of DiPerF and to ensure the desired scalability; the drawback of the small memory footprint per connection is the limited connection bandwidth that could be achieved per connection, but with 1,000s of concurrent connections, this would hardly be an issue.

In summary, DiPerF has been designed from the ground up with scalability, performance, flexibility, and accuracy as its target goal, and based on the results in this thesis, we believe this target goal has been achieved.

2.3 Contributions

The contributions of this thesis are two fold: 1) a detailed empirical performance analysis of various components of the Globus Toolkit along with a few grid services, and 2) DiPerF, a tool that makes automated distributed performance testing easy.

Through our tests performed on GRAM, WS-MDS, and GridFTP, we have been able to quantify the performance gain or loss between various different versions or implementations, and have normally found the upper limit on both scalability and performance on these services. We have also been able to show the performance of these components in a WAN, a task that would have been very tedious and time consuming without a tool such as DiPerF. By pushing the Globus Toolkit to the limit in both performance and scalability, we was able to give the users a rough overview of the performance they are to expect so they can do better resource planning. The developers also gained feedback on the behavior of the various components under heavy stress and allowed them to concentrate on improving the parts that needed the most improvements. We were also able to quantify the performance and scalability of DI-GRUBER, a distributed grid service built on top of the Globus Toolkit 3.2 and the Globus Toolkit 3.9.5.

The second main contribution is DiPerF itself, which provides a tool that allows large scale testing of grid services, web services, and network services to be done in both LAN and WAN environments. DiPerF has been automated to the extent that once configured, the framework will automatically do the following steps:

- check what machines or resources are available for testing
- deploy the client code on the available machines
- perform time synchronization
- run the client code in a controlled and predetermined fashion
- collect performance metrics from all the clients
- stop and clean up the client code from the remote resources
- aggregate the performance metrics at a central location
- summarize the results
- generates graphs depicting the aggregate performance of the clients and tested service

In summary, DiPerF offers an easy solution to large scale distributed performance measurements.

3 Related Work & Background Information

This section covers the related work to DiPerF including similar studies performed on grid services and various grid components. It also addresses basic and introductory information on various topics in order to make this thesis self contained.

3.1 Grid Services and Grid Performance Studies Related Work

We first cover related work for grid performance studies in general, studies on various components such as GridFTP, MDS, and GRAM, and finally the related work to the grid services we tested.

3.1.1 Grid Performance Studies

NetLogger [9] targets instrumentation of Grid middleware and applications, and attempts to control and adapt the amount of instrumentation data produced in order not to generate too much monitoring data. NetLogger is focusing on monitoring, and requires code modification in the clients; furthermore, it does not address automated client distribution or automatic data analysis.

GridBench [10] provides benchmarks for characterizing Grid resources and a framework for running these benchmarks and for collecting, archiving, and publishing results. While DiPerF focuses on performance exploration for entire services, GridBench uses synthetic benchmarks and aims to test specific functionalities of a Grid node. However, the results of these benchmarks alone are probably not enough to infer the performance of a particular service.

The development team of the Globus Toolkit have done extensive testing [11, 12] of the Globus Toolkit in LAN environments. Some of the tests they performed are even more involved and complex than what we have tested in this work, but the downside of these results is the artificial environment that is created in a LAN setup with multiple clients running on few machines. The results we obtained with 100s of machines distributed all over the world are much more likely to depict the realistic performance of the various Globus Toolkit components.

Grid applications can combine the use of compute, storage, network, and other resources. These resources are often geographically distributed, adding to application complexity and thus the difficulty of understanding application performance. GridMapper [13] is a tool for monitoring and visualizing the behavior of such distributed systems. GridMapper builds on basic mechanisms for registering, discovering, and accessing performance information sources, as well as for mapping from domain names to physical locations. The visualization system itself then supports the automatic layout of distributed sets of such sources and animation of their activities.

In grid computing environments, network bandwidth discovery and allocation is a serious issue. Before their applications are running, grid users will need to choose hosts based on available bandwidth. Running applications may need to adapt to a changing set of hosts. Hence, a tool is needed for monitoring network performance that is integral to the grid environment. To address this need, Gloperf [14] was developed as part of the Globus grid computing toolkit. Gloperf is designed for ease of deployment and makes simple, end-to-end TCP measurements requiring no special host permissions. Scalability is addressed by a hierarchy of measurements based on group membership and by limiting overhead to a small, acceptable, fixed percentage of the available bandwidth.

The Network Weather Service (NWS) [15] is a distributed monitoring and forecasting system. A distributed set of performance sensors feed forecasting modules. There are important differences to DiPerF. First, NWS does not attempt to control the offered load on the target service but merely to monitor it. Second, the performance testing framework deployed by DiPerF is built on the fly, and removed as soon as the test ends; while NWS sensors aim to monitor network performance over long periods of time.

3.1.2 GRAM Performance Studies Related Work

The Globus Toolkit's 3.2 job submission service test suite [16] uses multiple threads on a single node to submit an entire workload to the server. However, this approach does not gauge the impact of a wide-area environment, and does not scale well when clients use many resources, which means that the service will be relatively hard to saturate. The Globus Toolkit 3.9.4 job submission was also partially tested by the same group [12], but the tests are incomplete, and do not cover nearly the level of detail that the tests presented in this work.

3.2 Background Information on DiPerF components, test cases, and testbeds

This section contains a brief overview of various key concepts, testbeds, and software packages used in this work in order to make this thesis self contained. It covers 1) the definition of Grid Computing, 2) the description of four different testbeds (Grid3, PlanetLab, UChicago CS cluster, and the DiPerF cluster), 3) communication protocols (ssh, TCP, UDP), and 4) the background information on the various test cases (Globus Toolkit 3.2, 3.9.5, GRAM, GridFTP, WS-MDS, and GRUBER).

3.2.1 Testbeds

This section covers the four testbeds (Grid3, PlanetLab, the University of Chicago CS cluster, and the DiPerF cluster) that we used in this work. For each set of experiments in this work, we outline what testbed we used; this is an important section since the results of certain tests might vary with the particular testbed.

3.2.1.1 PlanetLab

PlanetLab [64] is a geographically distributed platform for deploying, evaluating, and accessing planetary-scale network services. PlanetLab is a shared community effort by a large international group of researchers, each of whom gets access to one or more isolated "slices" of PlanetLab's global resources via a concept called distributed virtualization. In order to encourage innovation in infrastructure, PlanetLab decouples the operating system running on each node from a set of multiple, possibly 3rd-party network-wide services that define PlanetLab, a principle referred to as unbundled management.

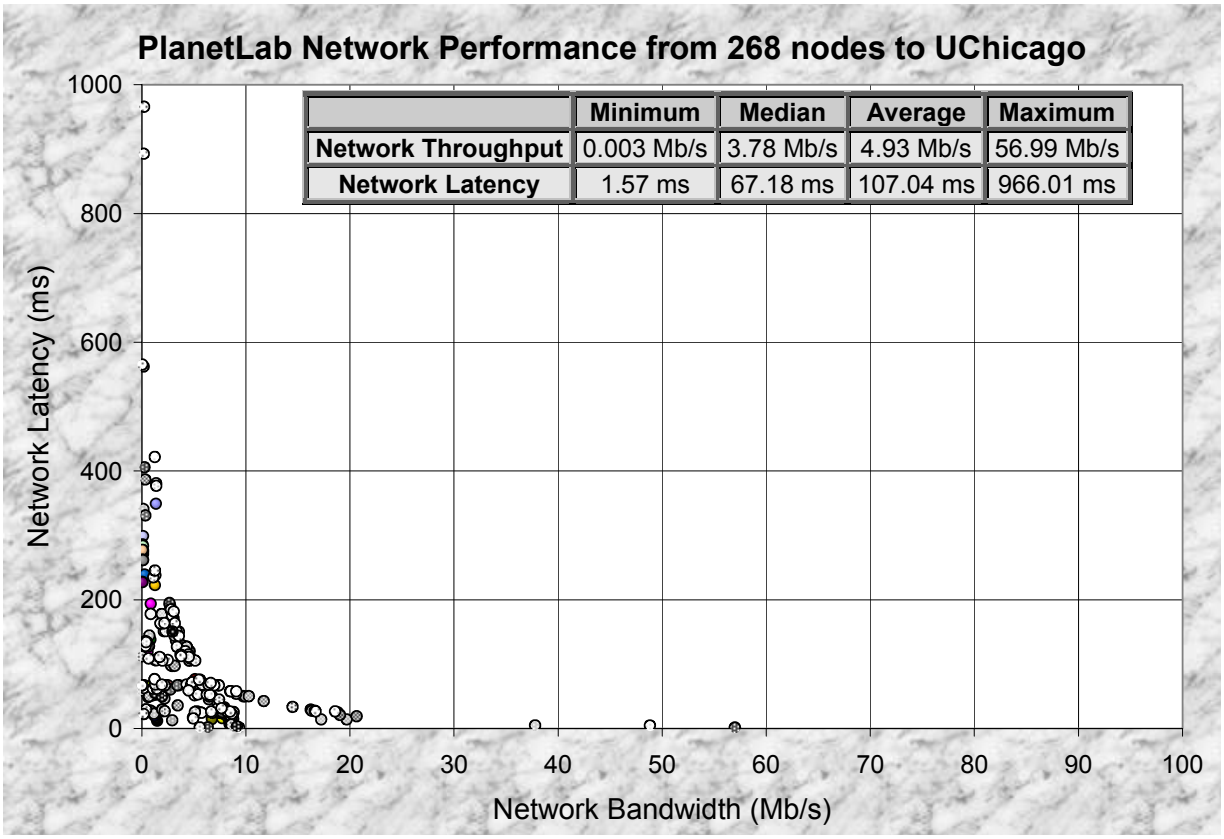

Figure 1: PlanetLab Network Performance from 268 nodes to a node at UChicago as measured by IPERF on April 13th, 2005; each circle denotes a physical machine with the corresponding x-axis and y-axis values as its network characteristics, namely network latency and bandwidth.

PlanetLab's deployment is now at over 500 nodes (Linux-based PCs or servers connected to the PlanetLab overlay network) distributed around the world. Almost all nodes in PlanetLab are connected via 10 Mb/s network links

(with 100Mb/s on several nodes), have processors speeds exceeding 1.0 GHz IA32 PIII class processor, and at least 512 MB RAM. Due to the large geographic distribution (the entire world) among PlanetLab nodes, network latencies and achieved bandwidth varies greatly from node to node. In order to capture this variation in network performance, Figure 1 displays the network performance of 268 nodes (the accessible nodes on 04-13-05) as measured by IPERF on April 13th, 2005. It is very interesting to note the heavy dependency between high bandwidth / low latencies and low bandwidth / high latencies. In order to visualize the majority of the node characteristics better, Figure 2 shows the same data from Figure 1, but with the x and y axis shown at log scale.

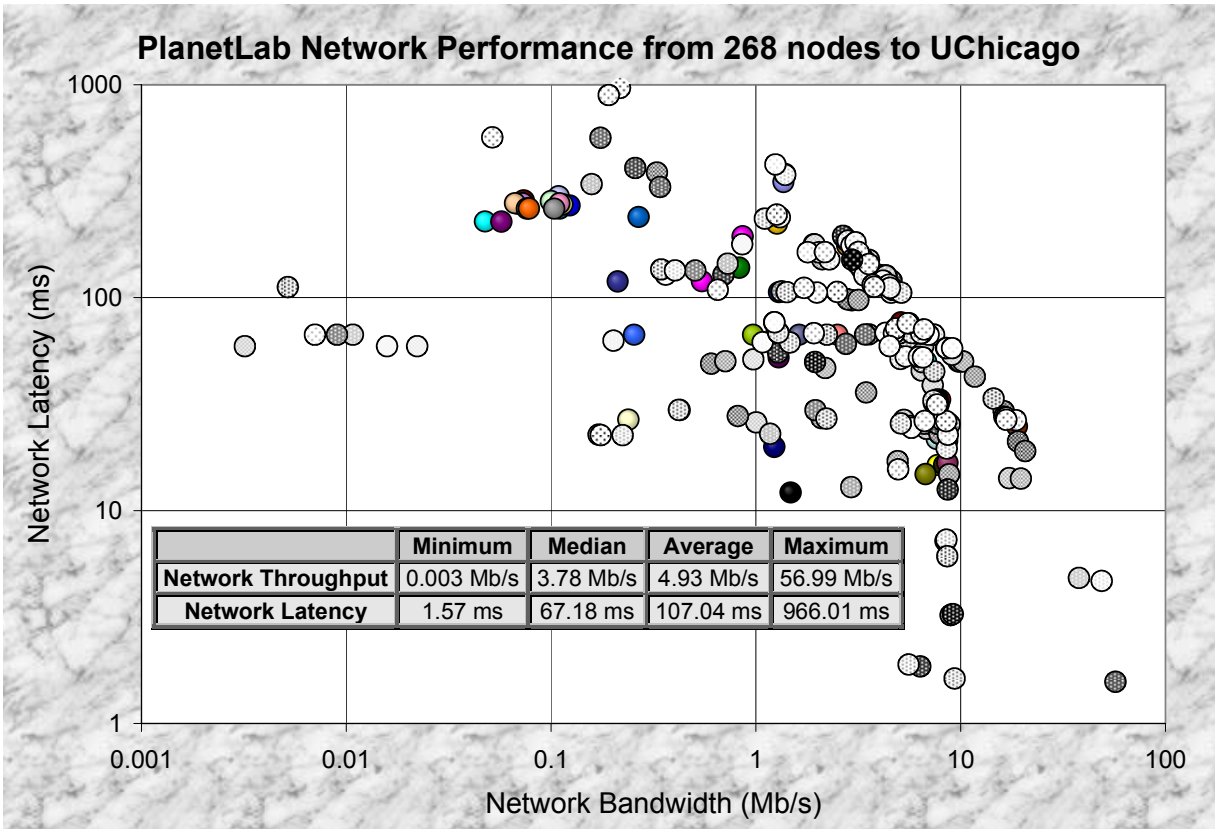

Figure 2: PlanetLab Network Performance from 268 nodes to a node at UChicago as measured by IPERF on April 13th, 2005 shown with x and y axis in log scale.

3.2.1.2 UChicago CS Cluster

The University of Chicago CS cluster contains over 100 machines that are remotely accessible. The majority of these machines are running Debian Linux 3.0, have AMD Athlon XP Processors at 2.1GHz, have 512 MB of RAM, and are connected via a 100 Mb/s Fast Ethernet switched network. The communication latency between any pair of machines in the CS cluster is on average less than 1 ms, with a few having latencies as high as several ms. Furthermore, all machines share a common file system via NFS (Network File System).

3.2.1.3 DiPerF Cluster

Some tests were performed on a smaller scale LAN that had better network connectivity, specifically 1Gb/s connections via a switch. The network latencies incurred were generally less than 0.1 ms. This cluster did not run NFS as was the case in the UChicago CS cluster. The connectivity of the DiPerF cluster to the outside world is 100Mb/s.

Table 1: DiPerF cluster at UChicago hosts hardware and OS details

Machine Name	m5	diablo	cobra	512tr	viper
Machine Type	x86 64 bit	x86 64 bit	X86 32 bit	X86 32 bit	X86 32 bit
OS	Linux Mandrake 10.1	Linux Suse 9.2	Linux Suse 9.2	Linux Suse 9.2	Linux Mandrake 10.1
OS Release	2.6.8.1-12mdksmp	2.6.8-24-default	2.6.4-52-default	2.6.8-24-default	2.6.8.1-12mdk
# of Proc.	2	1	1	1	1
CPU Speed	1600 MHz	1800 MHz	2166 MHz	2166 MHz	1466 MHz
Cache Size	L1: 256KB L2: 2048KB	L1: 128KB L2: 256KB	L1: 128KB L2: 256KB	L1: 128KB L2: 256KB	L1: 128KB L2: 256KB
CPU Type	AMD Opteron	AMD Athlon 64	AMD Athlon XP	AMD Athlon XP	AMD Athlon XP
Memory Total	1 GB DDR PC3200 Dual Channel	1 GB DDR PC3200 Dual Channel	1 GB DDR PC2700	1 GB DDR PC2700	768 MB SDRAM PC133
Swap Total	1 GB	1 GB	1 GB	1 GB	1 GB
Network Link	1 Gb/s Int. 100 Mb/s Ext.	1 Gb/s Int. 100 Mb/s Ext.	1 Gb/s Int. 100 Mb/s Ext.	1 Gb/s Int. 100 Mb/s Ext.	100 Mb/s Int. 100 Mb/s Ext.
Network MTU	1500 B	1500 B	1500 B	1500 B	1500 B

3.2.2 Test Cases

This section will cover the basics of the various components of the Globus Toolkit, including GridFTP, MDS, and GRAM. It will also cover the basic architecture of GRUBER, a grid service built on top of the Globus Toolkit 3.2.

3.2.2.1 Globus Toolkit 3.9.5

The Globus Toolkit 3.9.5 is in essence the Beta version of GT4 released in April 2005; since all the experiments were conducted prior to this date, all results are based on pre-GT4 releases. As shown in Figure 3, GT4 comprises both a set of service implementations ("server" code) and associated "client" libraries. GT4 provides both Web services (WS) components (on the top) and non-WS components (on the bottom). Note that all GT4 WS components use WS-Interoperability-compliant transport and security mechanisms, and can thus interoperate with each other and with other WS components. In addition, all GT4 components, both WS and non-WS, support X.509 end entity certificates and proxy certificates. Thus a client can use the same credentials to authenticate with any GT4 WS or non-WS component.

Nine GT4 services implement Web services (WS) interfaces:

- job management (GRAM)
- reliable file transfer (RFT)
- delegation
- Monitoring and Discovery System (MDS)

- MDS-Index
- MDS-Trigger
- MDS Aggregator

- community authorization (CAS)
- OGSA-DAI data access and integration
- GTCP Grid TeleControl Protocol for online control of instrumentation.

For two of these services, GRAM and MDS-Index, pre-WS "legacy" implementations are also provided. These pre-WS implementations will be deprecated at some future time as experience is gained with WS implementations. For three additional GT4 services, WS interfaces are not yet provided (but will be in the future):

- GridFTP data transport
- replica location service (RLS), and
- MyProxy online credential repository

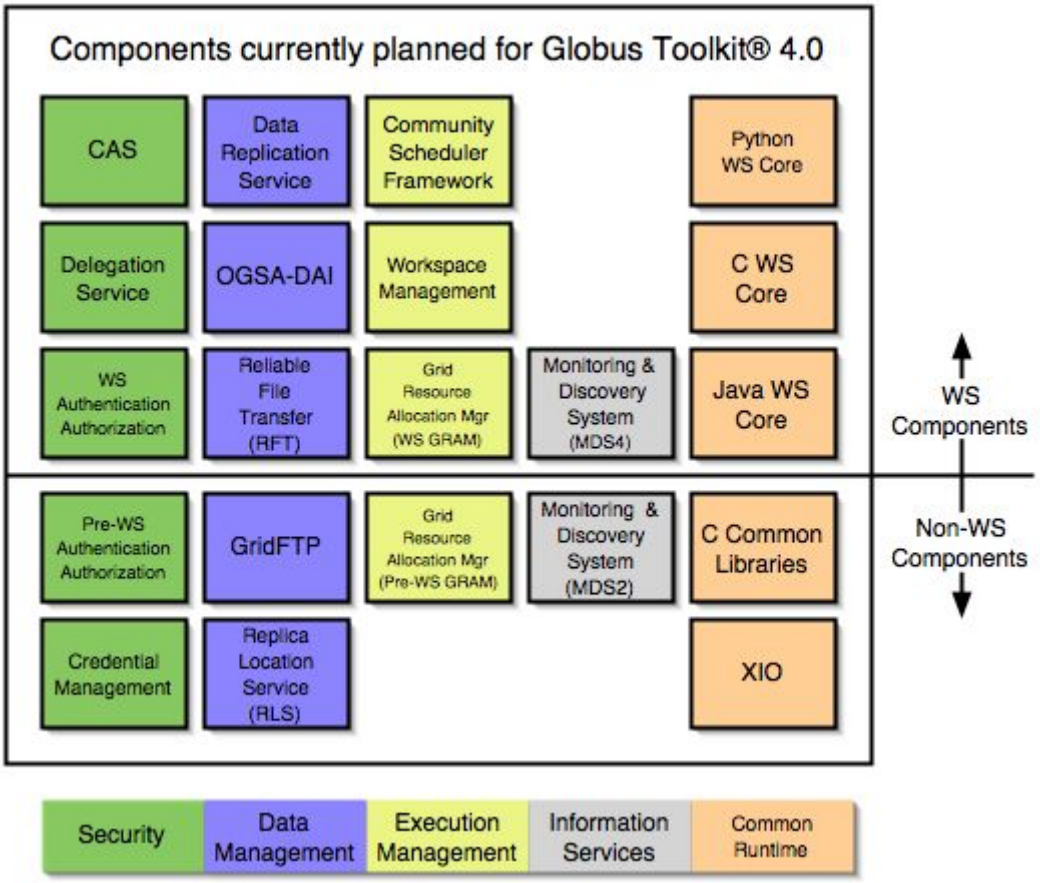

Figure 3: GT4 Key Components [73]

Other libraries provide powerful authentication and authorization mechanisms, while the eXtensible I/O (XIO) library provides convenient access to a variety of underlying transport protocols. SimpleCA is a lightweight certification authority.

3.2.2.2 GRAM

GRAM (Grid Resource Allocation and Management) simplifies the use of remote systems by providing a single standard interface for requesting and using remote system resources for the execution of "jobs". The most common use (and the best supported use) of GRAM is remote job submission and control. This is typically used to support distributed computing applications.

GRAM is designed to provide a single common protocol and API for requesting and using remote system resources, by providing a uniform, flexible interface to local job scheduling systems. The Grid Security Infrastructure (GSI) provides mutual authentication of both users and remote resources using GSI (Grid-wide) PKI-based identities. GRAM provides a simple authorization mechanism based on GSI identities and a mechanism to map GSI identities to local user accounts. [74]

We evaluated three implementations of a job submission service bundled with various versions of the Globus Toolkit:

- GT3.2 pre-WS GRAM
- GT3.2 WS GRAM
- GT4 WS GRAM.

GT3.2 pre-WS GRAM performs the following steps for job submission: a gatekeeper listens for job requests on a specific machine; performs mutual authentication by confirming the user's identity, and proving its identity to the user; starts a job manager process as the local user corresponding to authenticated remote user; then the job manager invokes the appropriate local site resource manager for job execution and maintains a HTTPS channel for information exchange with the remote user.

GT3.2 WS GRAM, a WS-based job submission service, performs the following steps: a client submits a *createService* request which is received by the Virtual Host Environment Redirector, which then attempts to forward the *createService* call to a User Hosting Environment (UHE) where mutual authentication / authorization can take place; if the UHE is not created, the Launch UHE module is invoked; WS GRAM then creates a new Managed Job Service (MJS); MJS submits the job into a back-end scheduling system [75].

GT3.9.5 WS-GRAM models jobs as lightweight WS-Resources rather than relatively heavyweight Grid services. WS GRAM combines job-management services and local system adapters with other service components of GT 4.0 in order to support job execution with coordinated file staging. Figure 4 depicts the complex set of message exchanges that occurs in WS-GRAM. We note that both pre-WS GRAM and WS GRAM are complex services: a job submission, execution, and result retrieval sequence may include multiple message exchanges between the submitting client and the service. [76]

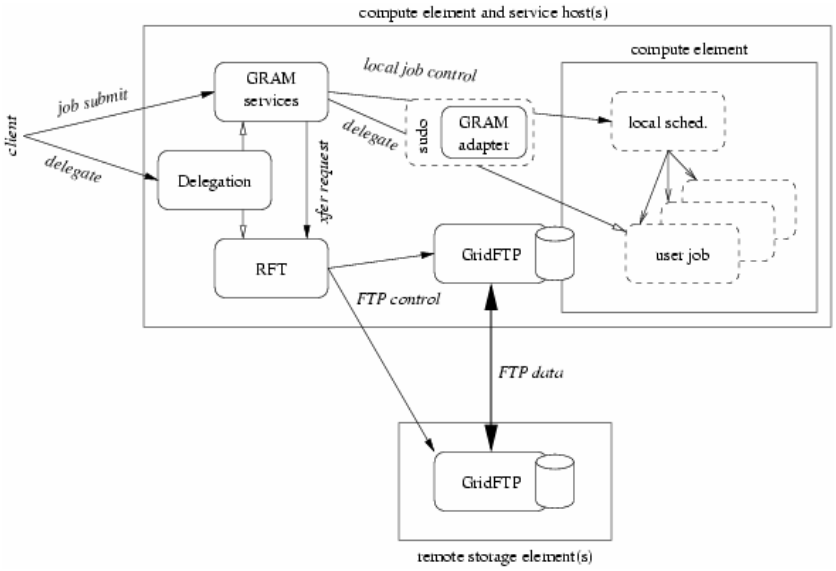

Figure 4: WS GRAM Component Architecture Approach [76]

The heart of the WS GRAM service architecture is a set of Web services designed to be hosted in the Globus Toolkit's WSRF core hosting environment. Each submitted job is exposed as a distinct resource qualifying the generic ManagedJob service. The service provides an interface to monitor the status of the job or to terminate the job (by terminating the ManagedJob resource). Each compute element, as accessed through a local scheduler, is exposed as a distinct resource qualifying the generic ManagedJobFactory service. The service provides an interface to create ManagedJob resources of the appropriate type in order to perform a job in that local scheduler.

At a high level, we can consider the main client activities around a WS GRAM job to be a partially ordered sequence as depicted in Figure 5. [76]

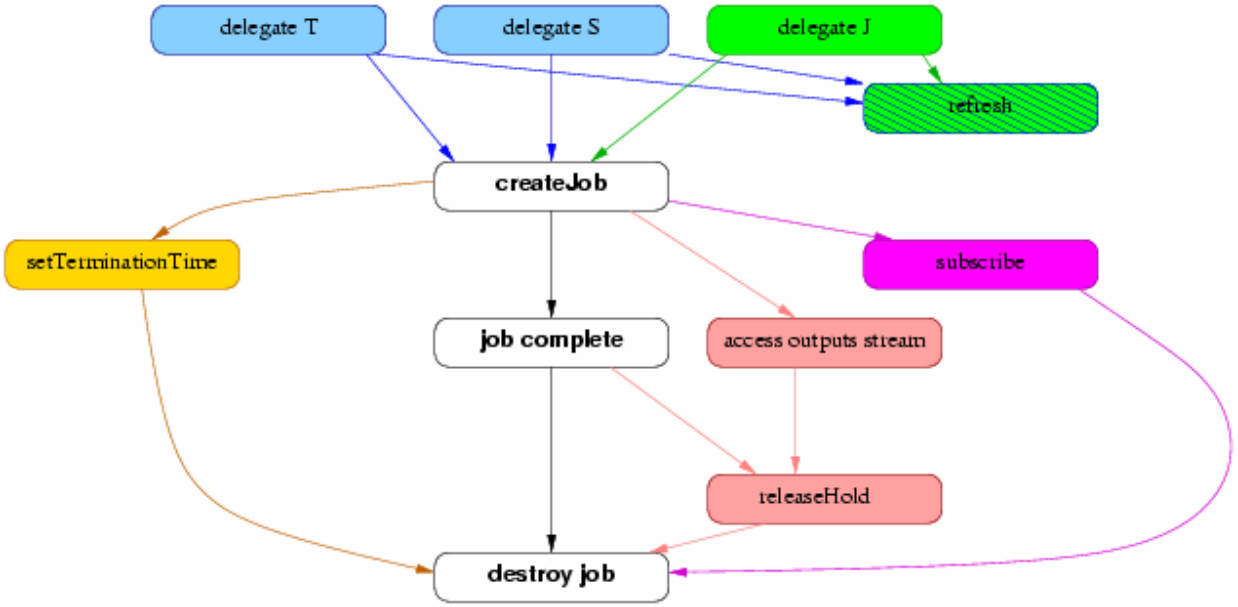

Figure 5: Partially ordered sequence of client activities around a WS GRAM job [76]

4 DiPerF Framework

DiPerF [82] is a DIstributed PERformance testing Framework aimed at simplifying and automating service performance evaluation. DiPerF coordinates a pool of machines that test a single or distributed target service, collects and aggregates performance metrics from the client point of view, and generates performance statistics. The aggregate data collected provides information on service throughput, service response time, on service 'fairness' when serving multiple clients concurrently, and on the impact of network latency on service performance. All steps involved in this process are automated, including dynamic deployment of a service and its clients, testing, data collection, and data analysis.

Figure 6 shows an example screenshot of a demo showing off the DiPerF from beginning to end. The screenshot below shows the end of the experiment in which the data analysis is complete and the graph with both the client and server view of the results have been generated; the GKrellM [83] Monitor is used to show the hosts (server and controller) performance in terms of CPU utilization, network transfer rates, and hard disk throughput.

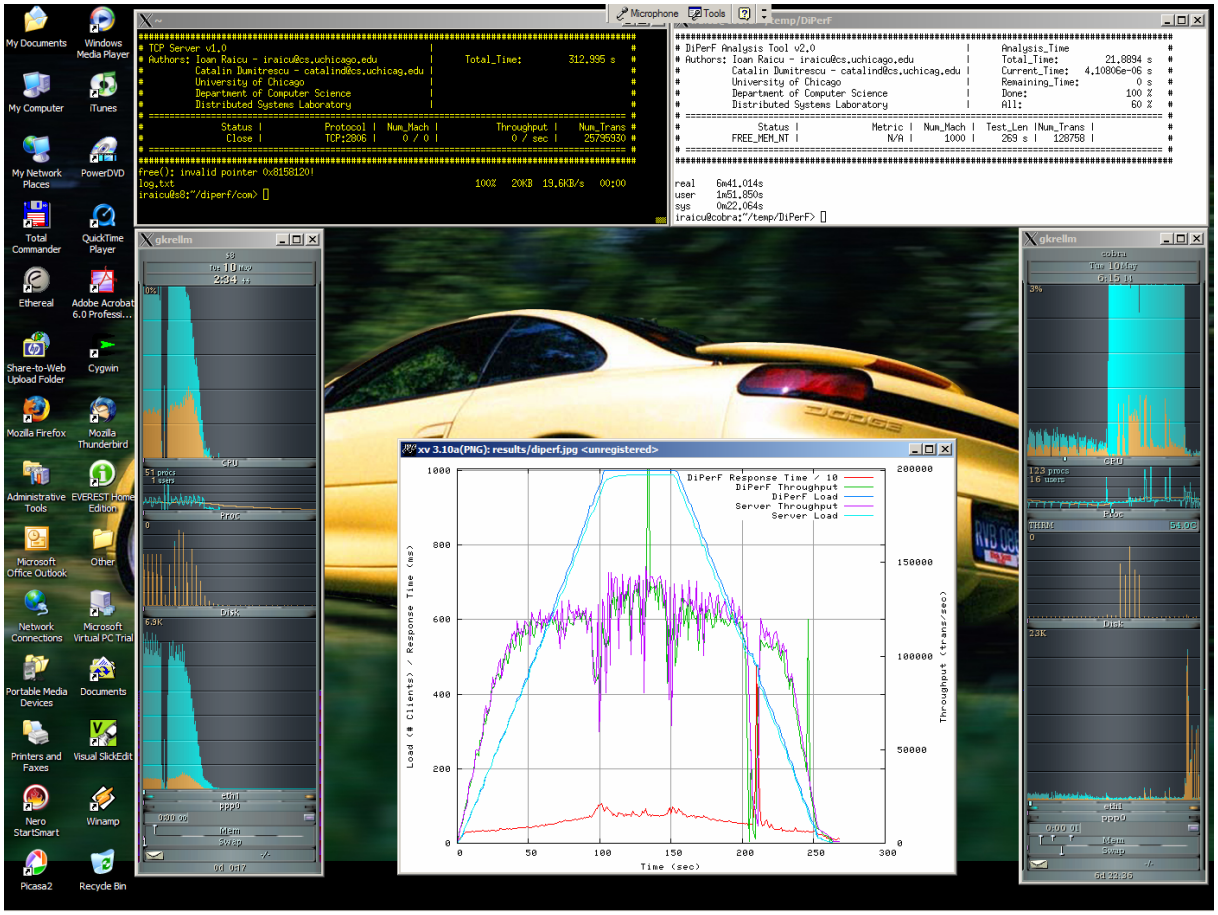

Figure 6: DiPerF demo showing the end of an experiment with 1000 clients accessing a TCP server

DiPerF consists of two major components: the *controller* and the *testers* (Figure 7). A user of the framework provides to the controller the address or addresses of the target service to be evaluated and the client code for the service. The controller starts and coordinates a performance evaluation experiment: it receives the client code, distributes it to testers, coordinates their activity, collects and finally aggregates their performance measurements. Each tester runs the client code on its machine, and times the (RPC-like) network calls this code makes to the target service. Finally, the controller collects all the measurements from the testers and performs additional operations (e.g., reconciling time stamps from various testers) to compute aggregated performance views.

Figure 8 depicts an overview of the deployment of DiPerF in different testbeds (PlanetLab, UChicago CS cluster, and Grid3). Note the different client deployment mechanisms between the different testbeds, with GT GRAM based submission for Grid3 and ssh based for the other testbeds. Another interesting difference between Grid3 and the other testbeds is the fact that the controller only communicates with a resource manager, and it is the resource manager's job to deploy and launch the tester/client code on physical machines in Grid3; in the other testbeds, the controller is directly responsible of having a complete list of all machines in the testbed and the communication is directly between the remote machine and the controller.

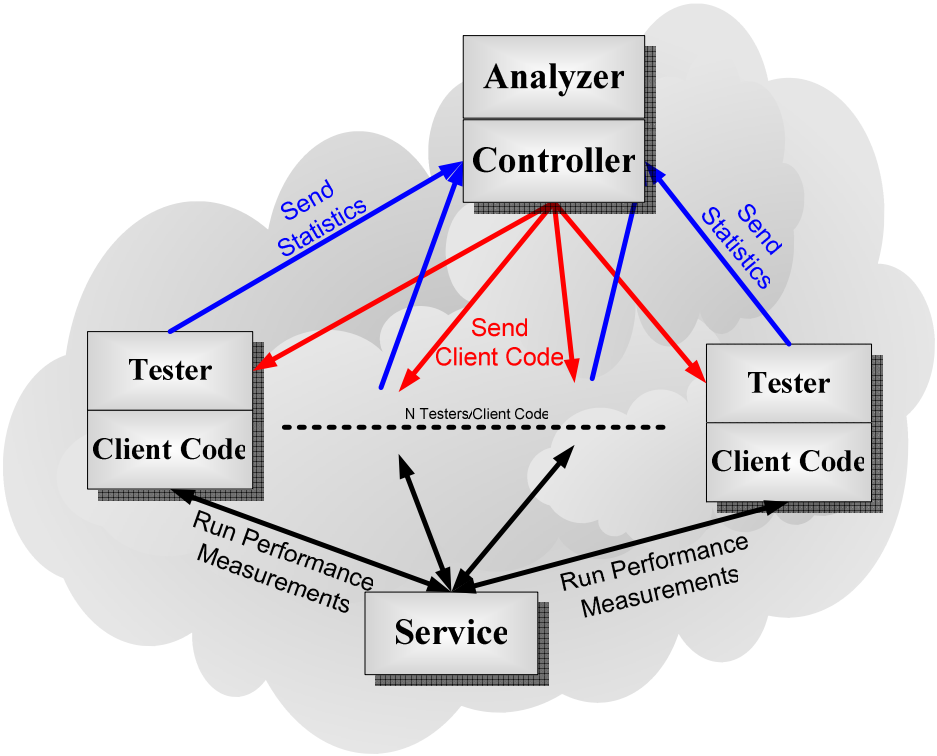

Figure 7: DiPerF framework overview

The interface between the tester and the client code can be defined in a number of ways (e.g., by using library calls); we take what we believe is the most generic avenue: clients are full blown executables that make one RPC-like call to the service. If multiple calls are to be made in each execution of the executable (i.e. when just starting the executable is expensive as is the case in many Java programs), a more sophisticated interface can be designed where the tester gets periodic information from the client code in a predefined format.

The framework is supplied with a set of candidate nodes for client placement, and selects those available as testers. In future work, we will extend the framework to select a subset of available tester nodes to satisfy specific requirements in terms of link bandwidth, latency, compute power, available memory, and/or processor load. In its current version, DiPerF assumes that the target service is already deployed and running.

Some metrics are collected directly by the testers (e.g., response time), while others are computed at the controller (e.g., throughput and service fairness). Additional metrics (e.g. network related metrics such as network throughput, size of data transmitted, time to complete a subtask, etc), measured by clients can be reported, through an additional interface, to the testers and eventually back to controller for statistical analysis. Testers send performance data to the controller while the test is progressing, and hence the service evolution and performance can be visualized 'on-line'.

Communication among DiPerF components has been implemented with several flavors: ssh based, TCP, and UDP. When a client fails, we rely on the underlying protocols (i.e. whatever the client uses such as TCP, UDP, HTTP, pre-WS GRAM, etc) to signal an error which is captured by the tester which is in turn sent to the controller to delete the client from the list of the performance metric reporters. A client could fail because of various reasons: 1) predefined timeout which the tester enforces, 2) client fails to start (i.e. out of memory - OS client machine related), 3) and service denied or service not found (service machine related). Once the tester is disconnected from the controller, it stops the testing process to avoid loading the target service with requests which will not be aggregated in the final results.

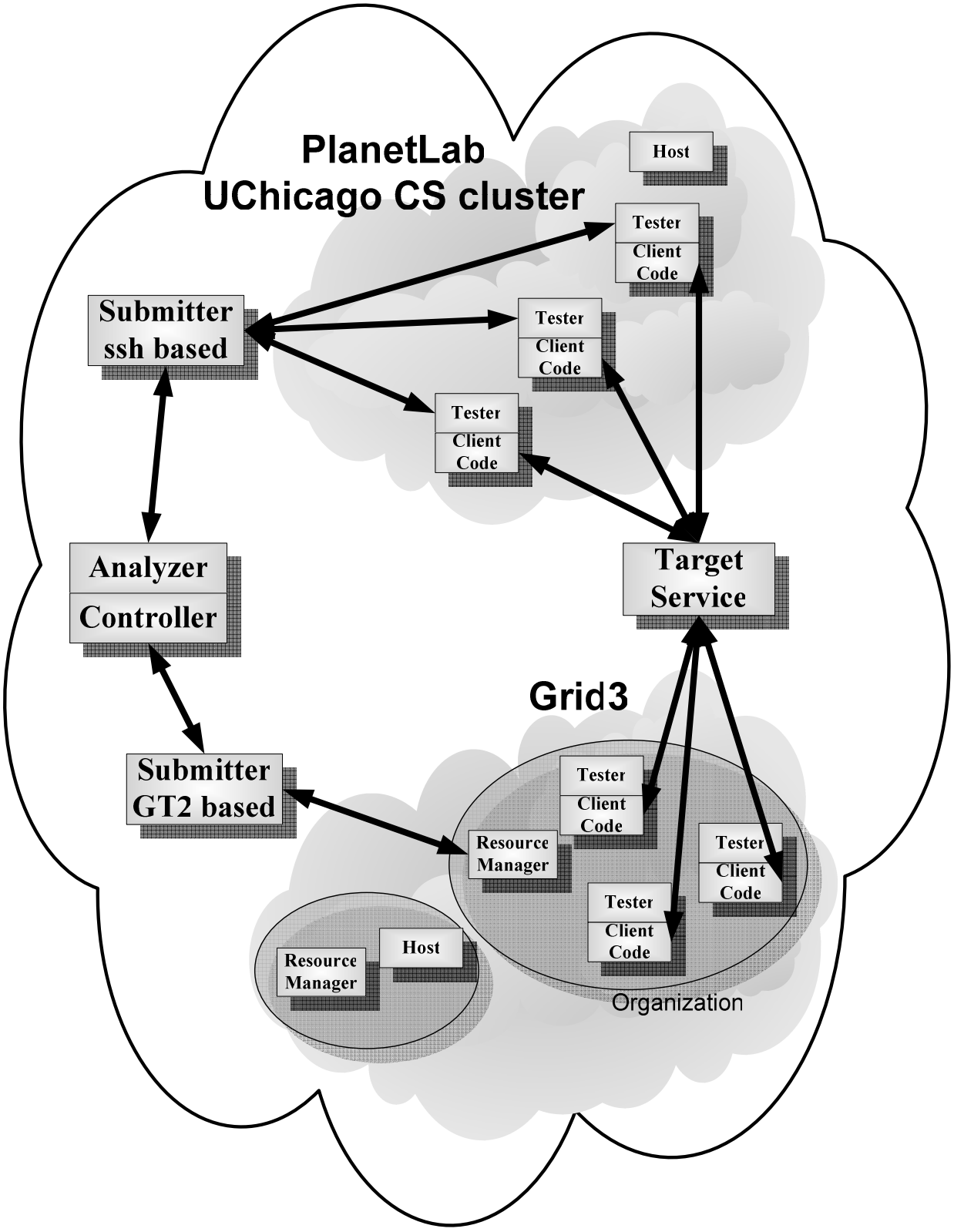

Figure 8: DiPerF framework overview deployment scenario

4.1 Scalability Issues

Our initial implementation [82] of DiPerF as it appeared at Grid2004 was scalable, but it could be improved, and hence we made several improvements to increase the scalability and performance of DiPerF. Based on the implementation using the ssh based communication protocol, DiPerF was limited to only about 500 clients. We therefore concentrated on reducing the amount of processing per transaction, reducing the memory footprint of the controller, and reducing the number of processes being spawned in relation to the number of desired testers/clients throughout the experiment.

In order to make DiPerF as flexible as possible for a wide range of configurations, we stripped down the controller from most of its data processing tasks (which are online) and moved them to the data analysis component (which is offline); this change helped the controller be more scalable, freeing the CPU of unnecessary load throughout the experiment. If it is desirable for the results to be viewed in real-time as the experiment progresses, then there exists another version of the controller that is more complex, but will give the user feedback of the performance in real-time. Furthermore, for increased flexibility, the controller can work in two modes: write data directly to the hard disk, or keep data in memory for faster analysis later and reduced load due to the fact that it does not have to write to the disk except when the experiment is over.

We also added the support of multiple testers on the same node by identifying a tester by the node name followed by its process ID (pid); this feature helped in managing multiple testers on the same node, which allows the testing of services beyond the size of the testbed. For example, if using a 100 physical node testbed, and having 10 testers/clients on each node, it would add up to 1000 testers/clients. This method of increasing the number of clients by running multiple clients on each physical node does not work for any client. If the client is heavy weight, and requires significant amounts of resources from the node, the average client performance will decrease as the number of clients increases; note that heavy weight clients will most likely produce inaccurate server performance once the number of clients surpasses the number of physical machines. On the other hand, this is a nice feature to have because it makes scalability studies for any client/service possible even with a small number of physical machines.

To alleviate the biggest bottleneck that we could identify, namely the communication based on ssh, we implemented two other communication protocols on top of TCP and UDP. Running TCP will allow us to have the same benefits of ssh (reliability), but will have less overhead because the information is not encrypted and decrypted, a relatively expensive operation. Using sockets, we can also control the buffer sizes of the connections more easily (without root privileges), and hence get better utilization of the memory of the system, especially since we can sacrifice buffer size without affecting performance due to the low needed bandwidth per connection. Using UDP, we get a stateless implementation, which will be much more scalable than any TCP or ssh based implementation. We get all the advantages of TCP (as mentioned above), but we loose the reliability; simple and relatively inexpensive methods to ensure some level of reliability could be implemented on top of UDP.

Finally, in order to achieve the best performance with the implementation of the communication over TCP or UDP, we used a single process which used the select() system function [84] to provide synchronous I/O multiplexing between 1,000s of concurrent connections. The functions select() wait for a number of file descriptors (found in fd_set) to change status based on a timeout value specified in the number of seconds and microseconds. First of all, the fd_set is a fixed size buffer as defined in several system header files; most Linux based systems have a fixed size of 1024. This means that any given select() function can only have 1024 file descriptors (i.e. TCP sockets) that it is listening on. This is a huge limitation, and would limit any 1 process implementation over TCP to only 1024 clients. After modifying some system files (required root access) to raise the constant size fd_set from 1024 to 65536, we were able to break the 1024 concurrent client barrier. However, we now had another issue to resolve, namely the expensive operation of initializing the fd_set (one file descriptor at a time) every time a complete pass through the entire fd_set; with an fd_set size of 1024, this did not seem to be a problem, but with an fd_set size of 65536, it quickly became a bottleneck. The solution we employed was to keep two copies of the fd_set, and after a complete pass through all the entire fd_set, simply do a memory to memory copy from one fd_set to another, a significantly less expensive operation than having to reset the fd_set one file descriptor at a time. We are currently working on porting the select()-based implementation to a /dev/poll-based [85] implementation that is considered to be significantly lighter weight than select() with less of an overhead.

With all these performance and scalability improvements, DiPerF can now scale from 4,000 clients using ssh to 60,000 clients using TCP to 80,000 clients using UDP; the achieved throughput varied from 1,000 to 230,000

transactions per second depending on the number concurrent clients and the communication protocol utilized. We expect DiPerF's scalability to increase even more once we replace the select() mechanism with /dev/poll; furthermore, we expect DiPerF's achieved throughput to increase even more under high concurrency with 10,000+ clients.

4.2 Client Code Distribution

The mechanisms used to distribute client code (e.g., scp, gsi-scp, or gass-server) vary with the deployment environment. Since ssh-family utilities are deployed on just about any Linux/Unix, we base our distribution system on scp-like tools. DiPerF specifically uses rsync to deploy client code in a Unix-like environment (i.e. PlanetLab, UChicago CS cluster, DiPerF cluster), and it uses GT2 GRAM job submission to deploy client code in a grid environment (i.e. Grid3).

4.3 Clock Synchronization

DiPerF relies heavily on time synchronization when aggregating results at the controller; therefore, an automatic time synchronization among all clients is integrated into DiPerF to ensure that the final results are accurate. Synchronization need not be performed on-line; instead, we can compute the offset between local and global time and apply that offset when analyzing aggregated metrics, assuming that the clock drift over the course of an experiment is not significant. The solution to clock drift is to perform time synchronization on-line at regular intervals that are short enough for the drift to be negligible.

Several off-the-shelf options are available to synchronize the time between machines; one example is NTP [86], which some PlanetLab nodes use to synchronize their time. In a previous study [86], hosts had a mean delay of 33ms, median 32ms, and a standard deviation 115ms from their peer hosts used to synchronize their time with NTP. These results seem very promising, but unfortunately, at the time that we performed our experiments, we found that most of the nodes in our testbed on PlanetLab were not very well synchronized, with some nodes having synchronization differences in the thousands of seconds. Therefore DiPerF assumes the worst: no clock synchronization mechanism is provided by the deployment platform. To ensure that a mechanism exists to synchronize time among all nodes within tens of milliseconds accuracy, we implemented a timer component that allows all nodes participating in an experiment to query for a 'global' time.

DiPerF handles time synchronization with a centralized time-stamp server that allows time mapping to a common base. The time-stamp server is lightweight and can easily handle 1000+ concurrent clients. In order to avoid clock drift, each client synchronize its clock every five minutes; due to the lightweight time server and relatively rare time synchronization (every 300 seconds), we estimate that it could handle 1,000,000+.

We have measured the latency from over 100 clients (deployed in the PlanetLab testbed) to our timestamp server at UChicago over a period of almost 2 hours. During this interval that the (per-node) latency in the network remained fairly constant and the majority of the clients had a network latency of less than 80ms. The accuracy of the synchronization mechanism we implemented is directly correlated with the network latency and its variance, and in the worst case (non-symmetrical network routes), the timer can be off by at most the network latency. Using our custom synchronization component, we observed a mean of 62ms, median 57ms, and a standard deviation 52ms for the time skew between nodes in our PlanetLab testbed. See Figure 1 for a visual representation of the network performance of the PlanetLab testbed.

Given that the response time of the relatively heavy weight services (i.e. GRAM, GridFTP, MDS, GRUBER) that have been evaluated in this paper are at least one order of magnitude larger, we believe the clock synchronization technique implemented does not distort the results presented.

4.4 Client Control and Performance Metric Aggregation

The controller starts each tester with a predefined delay (specified in a configuration file when the controller is started) in order to gradually build up the load on the service as can be visualized in Figure 9. A tester understands a simple description of the tests it has to perform. The controller sends test descriptions when it starts a tester. The most important description parameters are: the duration of the test experiment, the time interval between two concurrent client invocations, the time interval between two clock synchronizations, and the local command that has

to be invoked to run the client. The controller also specifies the addresses of the time synchronization server and the target service.

Individual testers collect service response times. The controller's job is to aggregate these service response times, correlate them with the offered load and with the start/stop time of each tester and infer service throughput, and service 'fairness' among concurrent clients.

Since all metrics collected share a global time-stamp, it becomes simple to combine all metrics in well defined time quanta (seconds, minutes, etc) to obtain an aggregate view of service performance at any level of detail that is coarser than the collected data, an essential feature for summarizing results containing millions of transactions over short time intervals. This data analysis is completely automated (including graph generation) at the user-specified time granularity.

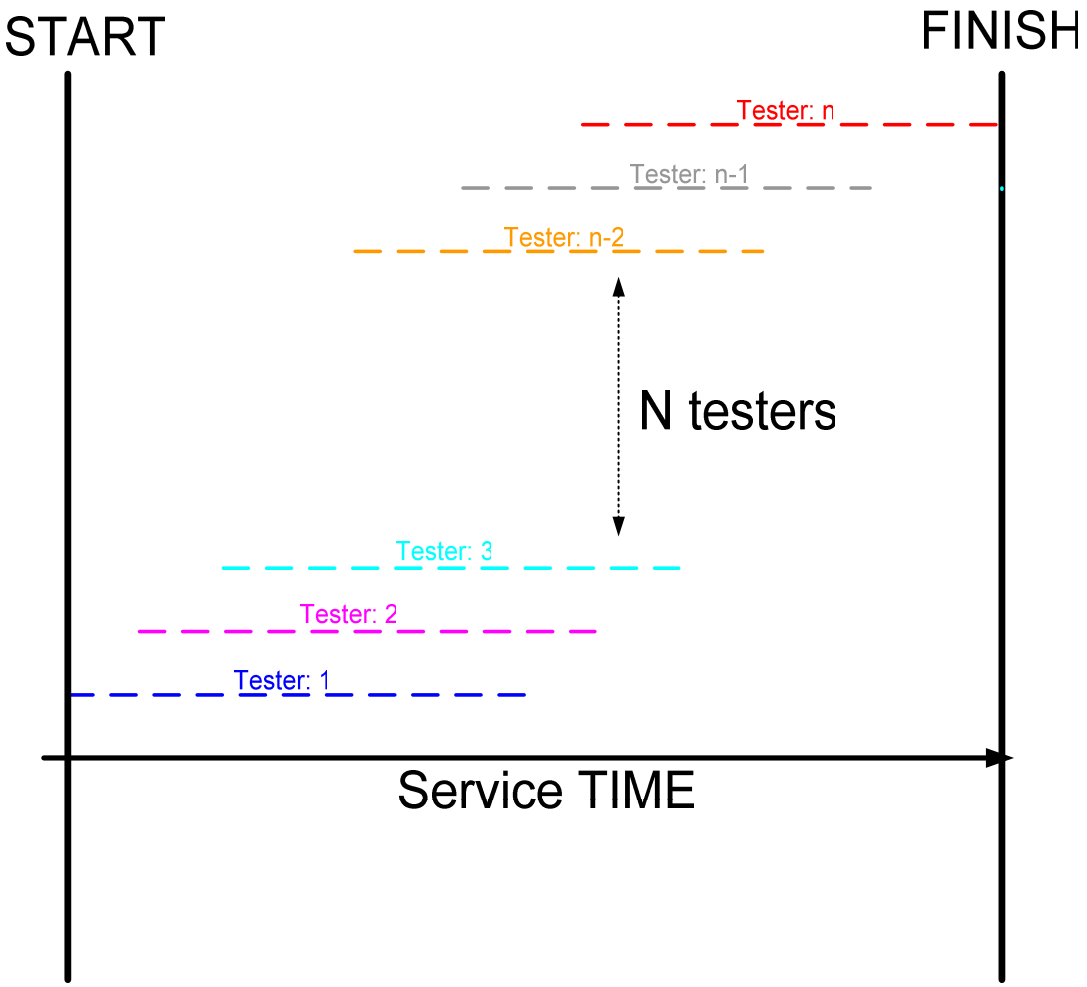

Figure 9: Aggregate view at the controller. Each tester synchronizes its clock with the time server every five minutes. The figure depicts an aggregate view of the controller of all concurrent testers.

4.5 Performance Analyzer

The performance analyzer has been implemented in C++ and currently consists of over 4,000 lines of code. Its current implementation assumes that all performance data is available for processing, which means it is an off-line process; in the future, we plan to port the current performance analyzer to support on-line analysis of incoming performance data. The implementation's main goal has been its flexibility in handling large data analysis tasks completely unsupervised. Furthermore, the performance analyzer was designed to allow a reduction of the raw

performance data to a summary of the performance data with samples computed at a specified time quantum. For example, a particular experiment could have accumulated 1,000,000s of performance samples over a period of 3600 seconds, but after the performance analyzer summarizes the data for one sample per second, the end result is reduced to 3600 samples rather than 1,000,000s of samples.

The generic metrics summarized by the performance analyzer based on the user specified time quantum are:

- *service response time* or time to serve a request, that is, the time from when a client issues a request to when the request is completed minus the network latency and minus the execution time of the client code; this metric is measured from the point of view of the client
- *service throughput*: number of jobs completed successfully by the service averaged over a short time interval that is specified by the user (i.e. 1 second, 1 minute, etc) in order to reduce the large number of samples; to make the results easier to understand, most of the graphs showing throughput also use box plot (moving averages) in order to smooth the throughput metrics out
- *offered load*: number of concurrent service requests (per second)
- *service utilization* (per client): ratio between the number of requests completed for a client and the total number of requests completed by the service during the time the client was active
- *service fairness* (per client): ratio between the number of jobs completed and service utilization
- *job success* (per client): the number of jobs that were successfully completed for a particular client
- *job fail* (per client): the number of jobs that failed for a particular client
- *network latency* (per client): the round trip network latency from the client to the service as measured by the "ping" utility

Among the many performance metrics it can extract from the raw performance data, it has a few additional features. First of all, it has a verify option that allows a user to double check that the raw input data conforms to the correct formatting requirements, and fixes (mostly by deleting) any inconsistencies it finds. Furthermore, all of the above metrics that are computed per client can also be computed over the peak portion of the experiment, when all clients are concurrently accessing the service. This is an important feature for computing the service fairness.

The output resulting from the performance analyzer can be automatically graphed using gnuplot [87] for ease of inspection, and the output can easily be manipulated in order to generate complex graphs combining various metrics such as the ones found in this thesis.

Figure 10 depicts a sample output from the automated gnuplot based on the summarization of the performance analyzer of the achieved throughput of a particular service. The original number of individual performance samples was over 75,000 while the resulting summary had fewer than 700 samples. On the other hand, Figure 11 shows the same graph manually made that contains the throughput (right axis) and two other metrics (load and response time) superimposed.

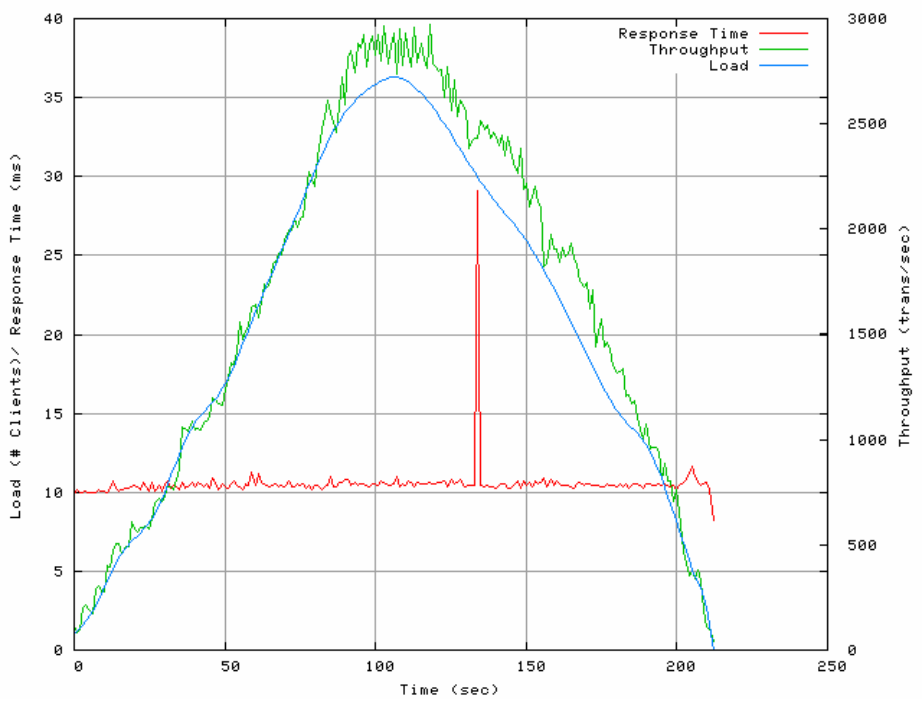

Figure 10: Sample output from the automated graph generated by gnuplot (throughput – right hand axis, response time and load – left hand axis)

For better presentation purposes, the majority of the results in this thesis will be presented similarly as the results from Figure 11. At this point, the actual results from these two figures is irrelevant, however the presentation and the kind of information that is expressed is very important.

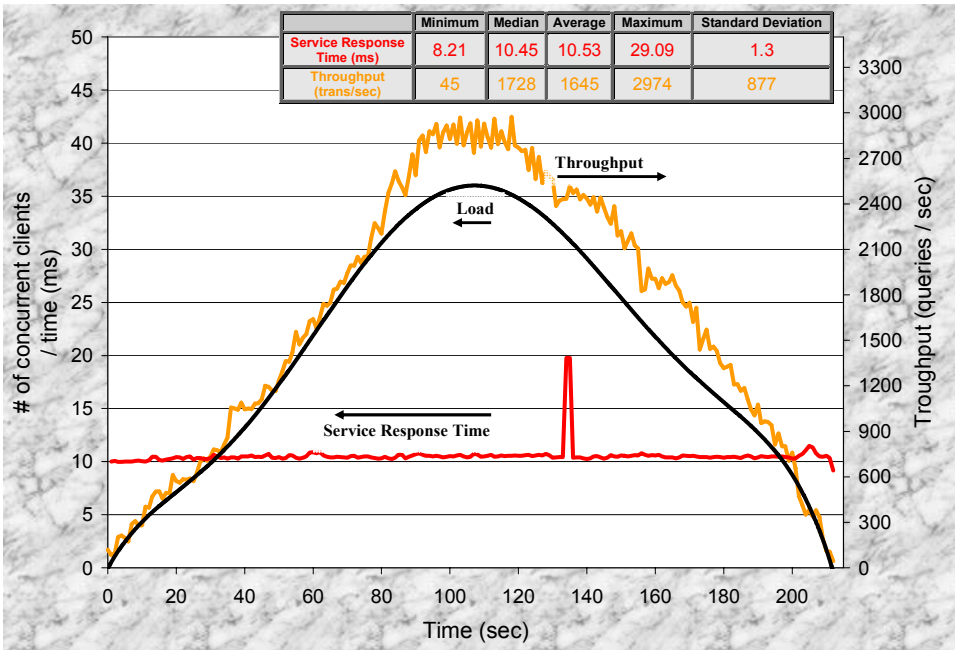

Figure 11: Sample output from the manual graph containing the same results (throughput – right hand axis) from Figure 10 plus two other metrics (response time and load)

5 Experimental Results for GRAM in GT3.2 & GT4

This section covers the experimental results obtained while studying various components of the Globus Toolkit (GRAM, WS-MDS, and GridFTP), and two grid services (DI-GRUBER, and a simple service that performed instance creation). For details on the specific services tested, please see section 2.3.

We ran our experiments with 89 client machines for pre-WS GRAM and 26 machines for WS GRAM, distributed over the PlanetLab testbed and the University of Chicago CS cluster (UofC). We ran the target services on an AMD K7 2.16GHz and the controller on an Intel PIII 600 MHz, both located at UofC. These machines are connected through 100Mbps Ethernet LANs to the Internet and the network traffic our tests generates is far from saturating the network links.

The actual configuration the controller passes to the testers is: testers start at 25s intervals and run for one hour during which they start clients at 1s intervals (or as soon as the last client completed its job if the time the client execution takes more than 1s). The client start interval is a tunable parameter, and is set based on the granularity of the service tested. In our case, since both services (pre-WS GRAM and WS GRAM) quickly rose to service response time of greater than 1s, for the majority of the experiments, testers were starting back-to-back clients. Experiments ran for a total of 5800s and 4200s for pre-WS GRAM and WS GRAM respectively. (The difference in total execution time comes from the different number of testers used). Testers synchronize their time every five minutes. The time-stamp server is another UofC computer.

For pre-WS GRAM, the tester input is a standalone executable that was run directly by the tester, while for the WS pre-WS GRAM, the input is a jar file and we assume that Java is installed on all testing machines in our testbed.

5.1 GT3.9.4 pre-WS GRAM and WS-GRAM Performance Results

We evaluated two implementations of a job submission service bundled with the Globus Toolkit 3.9.4:

- GT3.9.4 pre-WS GRAM (both client and service is implemented in C)
- GT3.9.4 WS GRAM (client is implemented in C while the service is in JAVA)

The metrics collected by DiPerF are:

- *service response time* or time to serve a request, that is, the time from when a client issues a request to when the request is completed minus the network latency and minus the execution time of the client code,
- *service throughput*: number of jobs completed successfully by the service averaged over a short time interval,
- *offered load*: number of concurrent service requests (per second),

We ran our experiments with 115 client machines or less for these experiments; the machines were distributed over the PlanetLab testbed. We ran the target services (GT3.9.4) on an AMD K7 2.16GHz and the controller on an identical machine, both located at UChicago. These machines are connected through 100Mbps Ethernet LANs to the Internet and the network traffic our tests generates is far from saturating the network links.

DiPerF was configured to start the testers at 25s intervals and run each tester for one hour during which they start clients at 1s intervals (or as soon as the last client completed its job if the time the client execution takes more than 1s). The client start interval is a tunable parameter, and is set based on the granularity of the service tested. In our case, since both services (pre-WS GRAM and WS GRAM) quickly rose to service response time of greater than 1s, for the majority of the experiments, testers were starting back-to-back clients.

Experiments ran anywhere from 100 seconds to 6500 seconds depending on how many clients were actually used. Testers synchronize their time every five minutes. The time-stamp server was another UChicago computer.

In the figures below, each series of points representing a particular metric and is also approximated using a moving average over a 60 point interval, where each graphs consists of anywhere from several thousand to several tens of thousands of data points.

5.1.1 WS-GRAM Results

Figure 12 depicts the performance of the WS-GRAM C client accessing the WS-GRAM service in JAVA. We note a dramatic improvement from the results of the WS-GRAM service implemented in the Globus Toolkit 3.2 as presented in **Error! Reference source not found.**. We observe both greater scalability (from 20 to 69 concurrent clients), and greater performance (from 10 jobs/min to over 60 jobs/minute). We also note that the response time steadily increased with the increased number of clients. The throughput increase seemed to level off at about 15~20 clients, which indicated that the service was saturated and that any more clients would only increase the response time.

Figure 13 shows a very similar experiment as the one showed in Figure 12 which justifies our choice of the number of concurrent clients to use in order to test the WS-GRAM service. Apparently, due to the alpha version of the GT3.9.4 release, there were still new features being added, software "bugs" to be fixed, and performance enhancements to be made. Unfortunately, through our relatively large scale experiments, we managed to trip a scalability problem in which the container would become unresponsive once we reached more than 70 concurrent clients. We therefore reran the experiment with only 69 clients, and obtained the results of Figure 12.

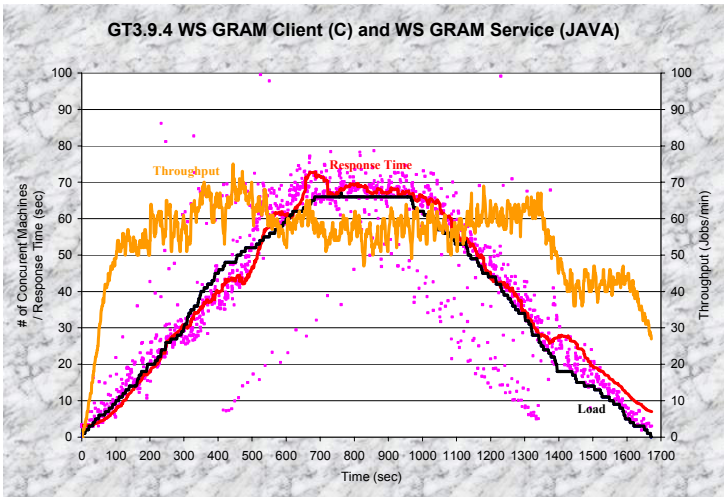

Figure 12: GT3.9.4 WS GRAM client (C implementation) and WS GRAM service (JAVA implementation); tunable parameters: utilized 69 concurrent nodes, starts a new node every 10 seconds, each node runs for 1000 seconds

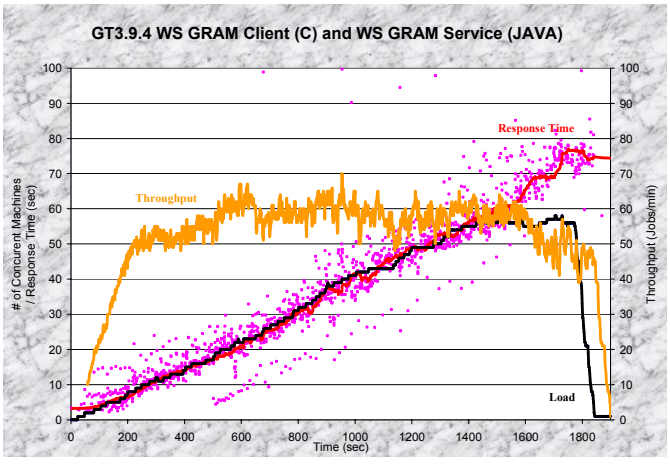

Figure 13: GT3.9.4 WS GRAM client (C implementation) and WS GRAM service (JAVA implementation); tunable parameters: attempted to use 115 concurrent nodes, but after 72 concurrent clients started, the

service became unresponsive; a new node was started every 25 seconds, and each node was scheduled to run for 3600 seconds

Figure 14 depicts the performance of the WS-GRAM client with the output enabled. We have enabled the remote client output to be sent back to the originating job submission point, however as can be seen in Figure 14, it seems that this incurs a big performance penalty over the same job submission mechanisms with the output disabled. We attempted to use 69 clients, however, at about concurrent 8 clients, the service became unresponsive.

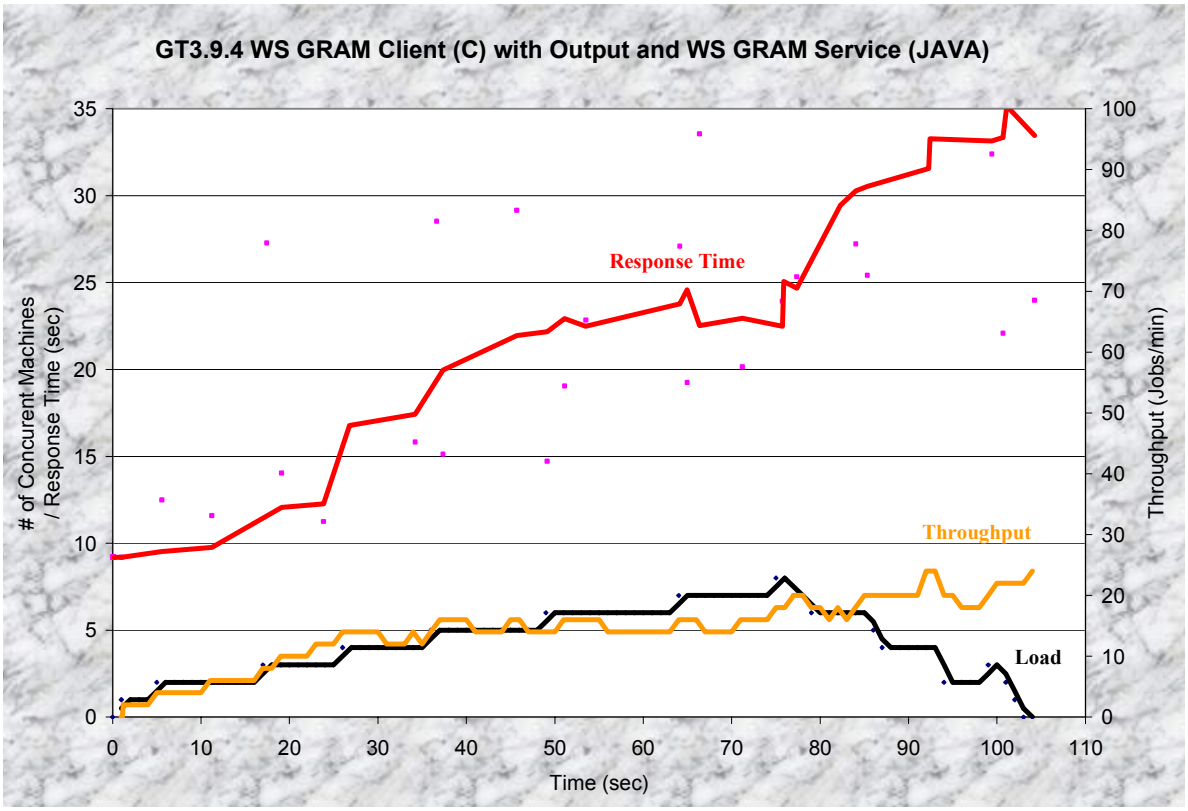

Figure 14: GT3.9.4 WS GRAM client (C implementation) with output enabled and WS GRAM service (JAVA implementation); tunable parameters: 8 concurrent clients, a new node was started every 10 seconds, and each node was scheduled to run for 1000 seconds

5.1.2 pre-WS GRAM Results

We performed a similar study on the older pre-WS GRAM implementation that is still bundled with GT3.9.4. We were interested to see how the performance of the pre-WS GRAM compared to that of WS-GRAM in the latest implementation of the Globus Toolkit; furthermore, we were also interested in finding out if enabling the output option had such an adverse effect on the pre-WS GRAM as it did in the WS-GRAM tests.

In comparing the results of the pre-WS GRAM (Figure 15) with those of the WS-GRAM service (Figure 12), we found very similar performance characteristics. The pre-WS GRAM seems to be more scalable, it withstood 115 clients vs. only 69 clients on the WS-GRAM service. Also, the response times seem to be a little less for the pre-WS GRAM service; it achieved service response times in the range of 50~60 seconds for 69 concurrent clients, while the WS-GRAM service achieved response times of close to 70 seconds. The throughout achieve by the pre-WS GRAM service also seems to be more consistent than that achieved by the WS-GRAM service. Based on our results, it seems that pre-WS GRAM with output performs only slightly worse than without output; there is about a 10% performance penalty, which is much lower than what we observed for the WS-GRAM. We see an achieved throughput of slightly less than 80 jobs/min without output and a throughput of slightly more than 70 jobs/min with output. The response times for about 60 clients increased from about 46 seconds to about 51 seconds.

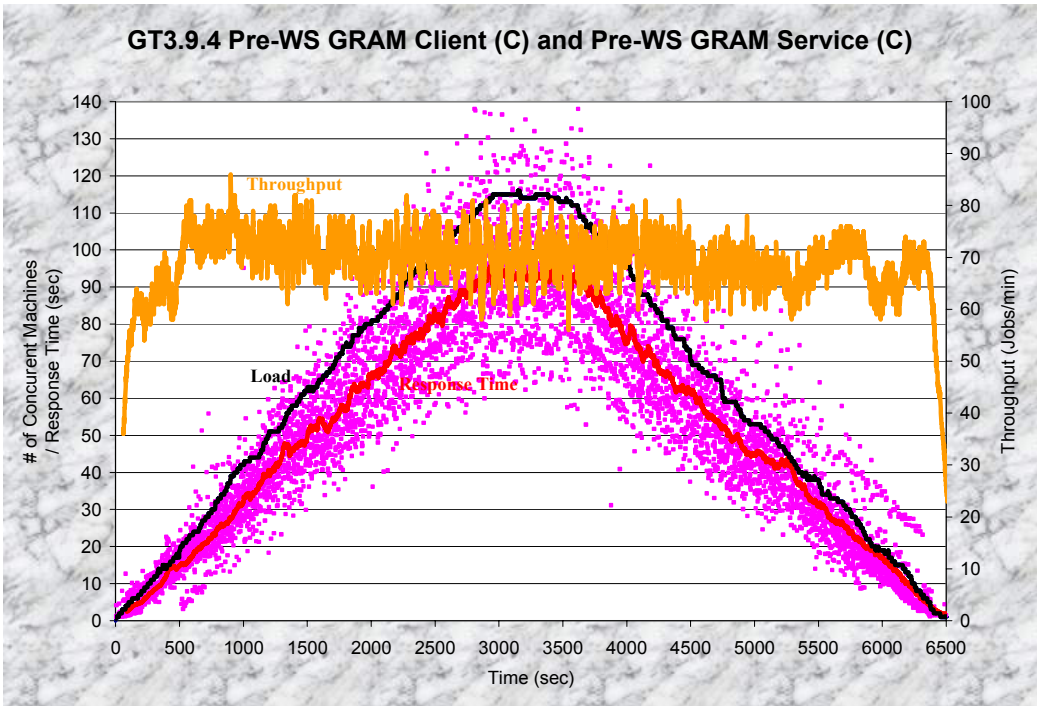

Figure 15: GT3.9.4 pre-WS GRAM client (C implementation) and pre-WS GRAM service (C implementation); tunable parameters: utilized 115 concurrent nodes, starts a new node every 25 seconds, each node runs for 3600 seconds

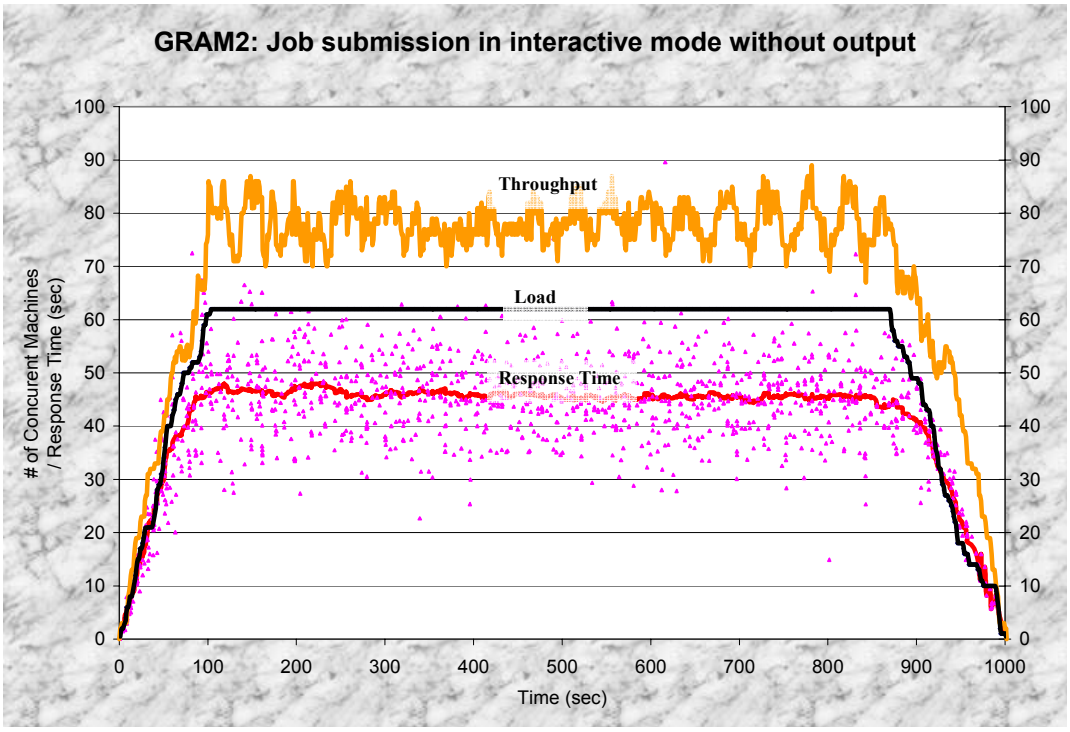

Figure 16: GRAM2 without output

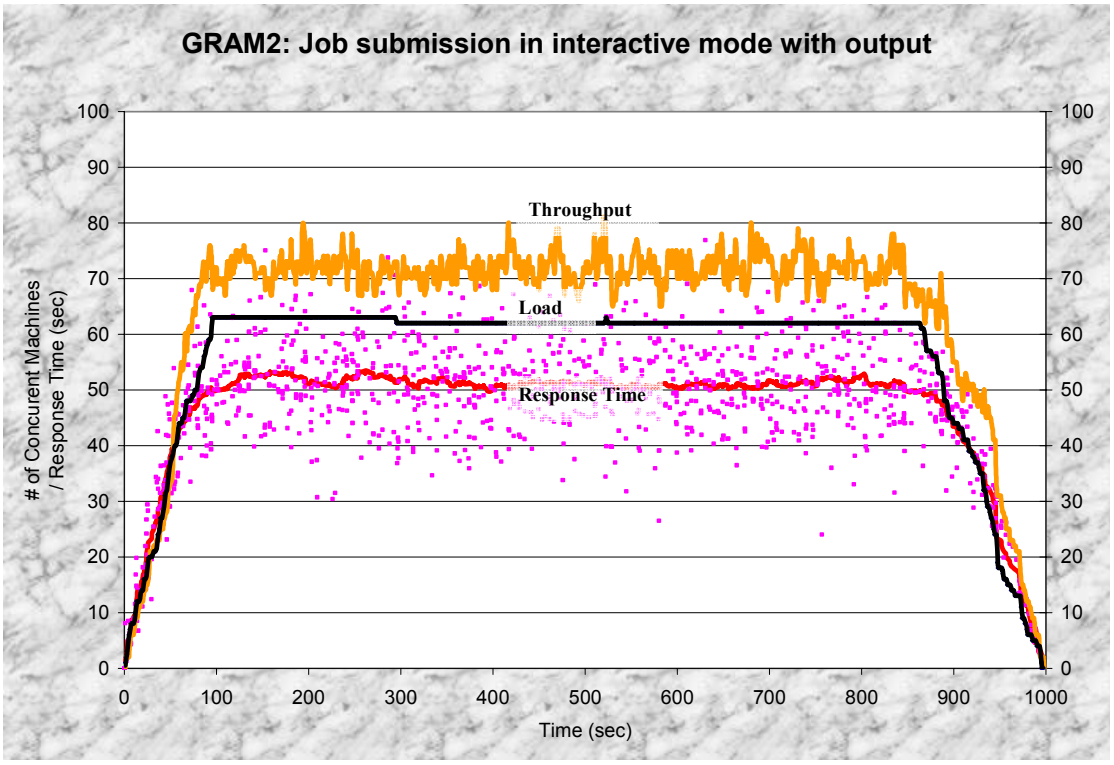

Figure 17: GRAM2 with output

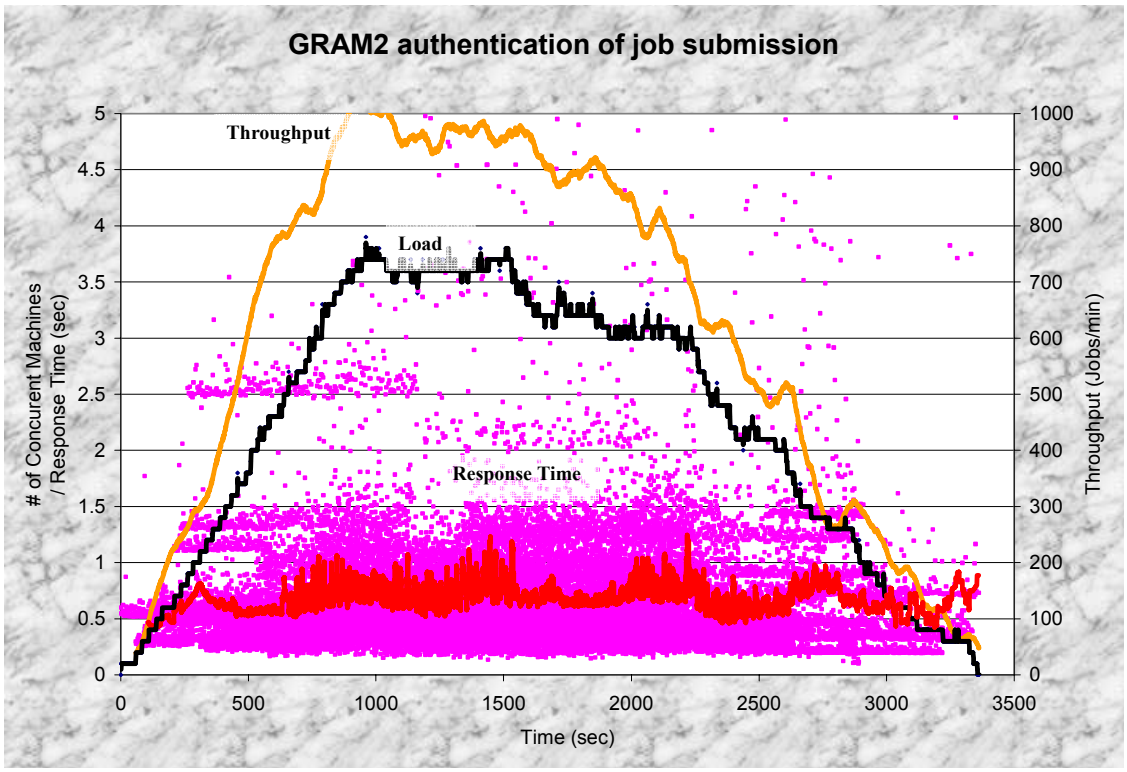

Figure 18: GRAM2 authentication

We also ran some tests to see how long the pre-WS GRAM authentication takes, and we found that it scales very well, at least to up to 38 concurrent clients. We actually had 120 clients in that test, but because of the test parameters (test duration and frequency of clients starting), it ended up that only about 38 clients were ever running concurrently. We see the throughput increase steadily the entire time that more clients join, up to a peak of over 1000 authentications per minute. It is also very interesting to see that the response time stayed relatively constant, between 0.5 seconds and 1 second, regardless of the number of concurrent clients.

5.1.3 GT3.9.4 GRAM Conclusions

Comparing the performance of pre-WS and WS GRAM, we find that pre-WS GRAM slightly outperforms WS GRAM by about 10% in both throughput and service response time. When we enabled the remote client output to be sent back to the originating job submission point, the performance of WS GRAM suffers tremendously. Also, for WS GRAM, if we loaded the service with too many concurrent clients, the service became unresponsive. We verified the performance of pre-WS GRAM with output and without output, and we found that the performance loss with output was only about 10%.

5.2 Summary

Table 2 summarizes the performance of pre-WS and WS-GRAM across both GT3 and GT4 found in the previous subsection. The experiments performed on GT3 and GT4 were done almost 1 year apart, and hence the results are not directly comparable between GT3 and GT4, but they should give a rough overview of the performance of both releases and implementation. It is noteworthy to point out the dramatic performance improvement WS-GRAM experienced from GT3 to GT4, going from less than 10 jobs per minute to almost 60 jobs per minute, and getting a reduction of response times from almost 180 seconds to less than 70 seconds.

Table 2: GRAM performance summary covering pre-WS GRAM and WS-GRAM under both GT3 and GT4

Experiment Description	Throughput (transactions/sec)					Load at Service Saturation	Response Time (sec)				
	Min	Med	Aver	Max	Std. Dev.		Min	Med	Aver	Max	Std. Dev.
Figure 27: GT3 pre-WS GRAM – 89 clients	99.3	193	193	326	38.9	33	1.02	20.4	31.5	739	41.7
Figure 30: GT3 WS-GRAM – 26 clients	3.13	9.16	8.77	12.8	2.19	20	35.6	178.4	173.6	298	55.9
Figure 36: GT4 pre-WS GRAM – 115 clients	57	69.8	69.8	81.6	4.23	27	57.6	92	93	167.2	15.1
Figure 33: GT4 WS-GRAM – 69 clients	47.2	57	56.8	63.9	3.1	20	32.1	67.7	67.4	131.8	9.8

5.3 Other Work that has used DiPerF

The papers or technical reports that used DiPerF along with the place of publishing are:

- DiPerF: an automated DIstributed PERformance testing Framework [82] – Grid2004
- A Scalability and Performance Evaluation of a distributed Usage SLA-based Broker in Large Grid Environments [26] – GriPhyN/iVDGL Technical Report, March 2005
- DI-GRUBER: A Distributed Approach for Grid Resource Brokering [25] – under review at SC 2005
- ZEBRA: The Globus Striped GridFTP Framework and Server [20] – under review
- Performance Measurements in Running Workloads over a Grid [27] – under review at SC 2005

- Extending a distributed usage SLA resource broker with overlay networks to support Large Dynamic Grid Environments [91] – work in progress
- Decreasing End-to-End Job Execution Times by Increasing Resource Utilization using Predictive Scheduling in the Grid [92] – UChicago, Grid Computing Seminar, 2005

Papers or technical reports that have mentioned DiPerF along with the place of publishing are:

- Systems Performance Evaluation Methods for Distributed Systems Using Datastreams [93] – MS Thesis, University of Kansas, January 2005
- Deploying C++ Grid Services: Options and Performance [94] – Duke University, Federated Distributed Systems, December 2004
- Connecting Client Objectives with Resource Capabilities: An Essential Component for Grid Service Management Infrastructures [95] – ICSOC 2004

6 Conclusion

As presented in this work, performing distributed measurements is not a trivial task, due to difficulties 1) *accuracy* – synchronizing the time across an entire system that might have large communication latencies, 2) *flexibility* – in heterogeneity normally found in WAN environments and the need to access large number of resources, 3) *scalability* – the coordination of large amounts of resources, and 4) *performance* – the need to process large number of transactions per second. In attempting to address these four issues, we developed DiPerF, a DIstributed PERformance testing Framework, aimed at simplifying and automating service performance evaluation. DiPerF coordinates a pool of machines that test a single or distributed target service (i.e. grid services, network services, distributed services, etc), collects and aggregates performance metrics (i.e. throughput, service response time, etc) from the client point of view, and generates performance statistics (fairness of resource utilization, saturation point of service, scalability of service, etc). The aggregate data collected provides information on service throughput, service response time, on service 'fairness' when serving multiple clients concurrently, and on the impact of network latency on service performance. Furthermore, using the collected data, it is possible to build predictive models that estimate service performance as a function of service load. Using the power of this framework, we have analyzed the performance scalability of the several components of the Globus Toolkit and several grid services. We measured the performance of various components of the GT in a wide area network (WAN) as well as a local area network (LAN) with the goal of understanding the performance that is to be expected from the GT in a realistic deployment in a distributed and heterogeneous environment.

The contributions of this thesis are two fold: 1) a detailed empirical performance analysis of various components of the Globus Toolkit along with a several grid services, and 2) DiPerF, a tool that makes automated distributed performance testing easy.

Through our tests performed on GRAM, WS-MDS, and GridFTP, we have been able to quantify the performance gain or loss between various different versions or implementations, and have normally found the upper limit on both scalability and performance on these services. We have also been able to show the performance of these components in a WAN, a task that would have been very tedious and time consuming without a tool such as DiPerF. By pushing the Globus Toolkit to the limit in both performance and scalability, we were able to give the users a rough overview of the performance they are to expect so they can do better resource planning. The developers also gained feedback on the behavior of the various components under heavy stress and allowed them to concentrate on improving the parts that needed the most improvements. We were also able to quantify the performance and scalability of DI-GRUBER, a distributed grid service built on top of the Globus Toolkit 3.2 and the Globus Toolkit 3.9.5.

The second main contribution is DiPerF itself, which is a tool that allows large scale testing of grid services, web services, network services, and distributed services to be done in both LAN and WAN environments. DiPerF has been automated to the extent that once configured, the framework will automatically do the following steps:

- check what machines or resources are available for testing
- deploy the client code on the available machines

- perform time synchronization
- run the client code in a controlled and predetermined fashion
- collect performance metrics from all the clients
- stop and clean up the client code from the remote resources
- aggregate the performance metrics at a central location
- summarize the results
- generates graphs depicting the aggregate performance of the clients and tested service

Some lessons we learned through the work presented in this thesis are:

- building scalable software is not a trivial task
    - there were some interesting issues we encountered when we tried to scale DiPerF beyond a few hundred clients, but after careful tuning and optimizations, we were able to scale DiPerF to 10,000+ clients
    - ssh is quite heavy weight for a communication channel, so for a real scalable solution, proprietary TCP/IP or UDP/IP communication channels are recommended
- time synchronization is a big issue when doing distributed measurements in which the aggregate view is important; although NTP offers potentially accurate clock synchronization, due to mis-configurations and possibly not wide enough deployment, NTP is not sufficient in a general case
- C-based components of the Globus Toolkit normally perform significantly better than their Java counterparts
- depending on the particular service tested, WAN performance is not always comparable to that found in a LAN; for example, WS-MDS with no security performed comparable between LAN and WAN tests, but WS-MDS with security enabled achieved less than half the throughput in a WAN when compared to the same test in a LAN
- the testbed performance (in our case it was mostly PlanetLab) can influence the performance results, and hence careful care must be taken in comparing experiments done at different times when the state of PlanetLab could have significantly changed

We conclude with the thought that we succeeded in building a scalable and high performance measurements tool that can be used to coordinate, measure, and aggregate the performance of thousands of clients distributed all over the world targeting anything from network services, web services, distributed services, to grid services. We have shown DiPerF's accuracy as being very good with only a few percent of performance deviation between the aggregate client view and the centralized service view. We have also contributed towards a better understanding of various vital Globus Toolkit components such as GRAM, MDS, and GridFTP.

7 Bibliography

[1] The Globus Alliance, www.globus.org.

[2] I. Foster, C. Kesselman "The Grid 2: Blueprint for a New Computing Infrastructure", "Chapter 1: Perspectives." Elsevier Publisher, 2003.

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

[4] I. Foster, C. Kesselman, J. Nick, S. Tuecke. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. Open Grid Service Infrastructure WG, Global Grid Forum, June 22, 2002.

[5] The Globus Alliance, "WS GRAM: Developer's Guide", http://www-unix.globus.org/toolkit/docs/3.2/gram/ws.

[6] L. Peterson, T. Anderson, D. Culler, and T. Roscoe, "A Blueprint for Introducing Disruptive Technology into the Internet", Proceedings of the First ACM Workshop on Hot Topics in Networking (HotNets), October 2002.

[7] A. Bavier et al., "Operating System Support for Planetary-Scale Services", Proceedings of the First Symposium on Network Systems Design and Implementation (NSDI), March 2004.

[8] I. Foster, et al., "The Grid2003 Production Grid: Principles and Practice", 13th IEEE Intl. Symposium on High Performance Distributed Computing, 2004.

[9] D. Gunter, B. Tierney, C. E. Tull, V. Virmani, On-Demand Grid Application Tuning and Debugging with the NetLogger Activation Service, 4th International Workshop on Grid Computing, Grid2003, Phoenix, Arizona, November 17th, 2003.

[10] G. Tsouloupas, M. Dikaiakos. "GridBench: A Tool for Benchmarking Grids," 4th International Workshop on Grid Computing, Grid2003, Phoenix, Arizona, November 17th, 2003.

[11] The Globus Alliance, "Globus Toolkit 3.0 Test Results Page", http://www-unix.globus.org/ogsa/tests/gt3_tests_result.html

[12] The Globus Alliance, "Overview and Status of Current GT Performance Studies", http://www-unix.globus.org/toolkit/docs/development/3.9.5/perf_overview.html

[13] W. Allcock, J. Bester, J. Bresnahan, I. Foster, J. Gawor, J. A. Insley, J. M. Link, and M. E. Papka. "GridMapper, A Tool for Visualizing the Behavior of Large-Scale Distributed Systems," 11th IEEE International Symposium on High Performance Distributed Computing (HPDC-11), pp179-187, Edinburgh, Scotland, July 24-16, 2002.

[14] C. Lee, R. Wolski, I. Foster, C. Kesselman, J. Stepanek. "A Network Performance Tool for Grid Environments," Supercomputing '99, 1999.

[15] R. Wolski, N. Spring, J. Hayes, "The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing," Future Generation Computing Systems, 1999.

[16] The Globus Alliance, "GT3 GRAM Tests Pages", http://www-unix.globus.org/ogsa/tests/gram.

[17] X. Zhang, J. Freschl, and J. Schopf. "A Performance Study of Monitoring and Information Services for Distributed Systems." Proceedings of HPDC, August 2003.

[18] X. Zhang and J. Schopf. "Performance Analysis of the Globus Toolkit Monitoring and Discovery Service, MDS2." Proceedings of the International Workshop on Middleware Performance (MP 2004), April 2004.

[19] G. Aloisio, M. Cafaro, I. Epicoco, and S. Fiore, "Analysis of the Globus Toolkit Grid Information Service". Technical report GridLab-10-D.1-0001-GIS_Analysis, GridLab project, http://www.gridlab.org/Resources/Deliverables/D10.1.pdf.

[20] B. Allcock, J. Bresnahan, R. Kettimuthu, M. Link, C. Dumitrescu, I. Raicu, I. Foster. "Zebra: The Globus Striped GridFTP Framework and Server", submitted for review.

[21] G. Kola, T. Kosar, and M. Livny. "Profiling Grid Data Transfer Protocols and Servers", Proceedings of Euro-Par 2004, September 2004.

[22] T. Baer, P. Wyckoff. "A Parallel I/O Mechanism for Distributed Systems", Proceedings of Cluster '04, San Diego, CA, September 2004.

[23] X. Liu, H. Xia, and A.A. Chien, "Network Emulation Tools for Modeling Grid Behavior," 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2003), May 12-15, 2003 in Tokyo, Japan.

[24] C. Dumitrescu, I. Foster. "GRUBER: A Grid Resource SLA-based Broker", EuroPar 2005.

[25] C. Dumitrescu, I. Raicu, I. Foster. "DI-GRUBER: A Distributed Approach for Grid Resource Brokering", submitted for review to SC 2005.

[26] C. Dumitrescu, I. Foster, I. Raicu. "A Scalability and Performance Evaluation of a distributed Usage SLA-based Broker in Large Grid Environments", GriPhyN/iVDGL Technical Report, March 2005.

[27] C. Dumitrescu, I. Raicu, I. Foster. "Performance Measurements in Running Workloads over a Grid", submitted for review to SC 2005.

[28] A. Adams, J. Mahdavi, M. Mathis, and V. Paxson. Creating a scalable architecture for Internet measurement. IEEE Network, 1998.

[29] V. Paxson, J. Mahdavi, A. Adams, and M. Mathis. An architecture for large-scale internet measurement. IEEE Communications, 36(8):48–54, August 1998.

[30] V. Paxson. End-to-end Internet packet dynamics. In Proceedings of ACM SIGCOMM '97, Cannes, France, September 1997.

[31] N. Anerousis, R. C´aceres, N. Duffield, A. Feldmann, A. Greenberg, C. Kalmanek, P. Mishra, K. Ramakrishnan, and J. Rexford. Using the AT&T labs PacketScope for Internet measurement. AT&T Services and Infrastructure Performance Symposium, November 1997.

[32] A. Feldmann. Continuous on-line extraction of HTTP traces from packet traces. Position paper: W3C Web Characterization Workshop, November 1998.

[33] Internet Protocol Performance Metrics. http://www.advanced.org/ippm/index.html, 1998.

[34] V. Paxson, G. Almes, J. Mahdavi, and M. Mathis. Framework for IP performance metrics. IETF RFC 2330, 1998.

[35] The Surveyor Project. http://www.advanced.org/csgippm/, 1998.

[36] National Laboratory for Applied Network Research. http://www.nlanr.net, 1998.

[37] NLANR Acitve Measurement Program - AMP. http://moat.nlanr.net/AMP.

[38] NLANR Passive Measurement and Analysis - PMA. http://moat.nlanr.net/PMA.

[39] A Distributed Testbed for National Information Provisioning. http://ircache.nlanr.net/Cache/.

[40] Keynote Systems Inc. http://www.keynote.com, 1998.

[41] Collaborative Advanced Interagency Research Network. http://www.cairn.net.

[42] Cooperative Association for Internet Data Analysis. http://www.caida.org, 1998.

[43] CAIDA. "Cooperative Association for Internet Data Analysis." CAIDA, 1996.

[44] Skitter. http://www.caida.org/tools/measurement/skitter.

[45] V. Jacobson. traceroute. ftp://ftp.ee.lbl.gov/traceroute.tar.Z, 1989.

[46] Coral: Passive network traffic monitoring and statistics collection. http://www.caida.org/tools/coral.

[47] C.R. Simpson Jr., G.F. Riley: NETI@home: A Distributed Approach to Collecting End-to-End Network Performance Measurements. PAM 2004: 168-174

[48] Ch. Steigner and J. Wilke, "Isolating Performance Bottlenecks in Network Applications", in Proceedings of the International IPSI-2003 Conference, Sveti Stefan, Montenegro, October 4-11, 2003.

[49] B.B. Lowekamp, N. Miller, R. Karrer, T. Gross, and P. Steenkiste, "Design, Implementation, and Evaluation of the Remos Network Monitoring System," Journal of Grid Computing 1(1):75--93, 2003.

[50] IEPM. Internet End-to-End and Process Monitoring (SLAC/DOE). http://wwwiepm.slac.stanford.edu/

[51] MAWI (WIDE Project), http://www.wide.ad.jp/wg/mawi/

[52] PPNCG Network Monitoring, http://icfamon.rl.ac.uk/ppncg/main.html

[53] TRIUMF Network Monitoring, http://sitka.triumf.ca/

[54] WAND (Waikato Applied Network Dynamics) WITS (Waikato Internet Traffic Storage) Project, http://wand.cs.waikato.ac.nz/

[55] Andover News Network's Internet Traffic Report, http://www.internettrafficreport.com/main.htm

[56] MIDS Internet Average, http://**average**.miq.net/

[57] MIDS Internet Weather Report, http://www.mids.org/weather/

[58] MIDS Matrix IQ Ratings Comparing Performance of Some ISPs, http://ratings.miq.net/

[59] NetSizer (Telcordia Technologies), http://www.netsizer.com/

[60] P. Barford ME Crovella. Measuring Web performance in the wide area. Performance Evaluation Review, Special Issue on Network Traffic Measurement and Workload Characterization, August 1999.

[61] G. Banga and P. Druschel. Measuring the capacity of a Web server under realistic loads. World Wide Web Journal (Special Issue on World Wide Web Characterization and Performance Evaluation), 1999.

[62] I. Foster and C. Kesselman, Eds., "The Grid: Blueprint for a Future Computing Infrastructure", "Chapter 2: Computational Grids." Morgan Kaufmann Publishers, 1999.

[63] Grid3. http://www.ivdgl.org/grid3/

[64] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman, "PlanetLab: An Overlay Testbed for Broad-Coverage Services," ACM Computer Communications Review, vol. 33, no. 3, July 2003.

[65] Information Sciences Institute, University of Southern California, "Internet Protocol," Request for Comments 791, Internet Engineering Task Force, September 1981.

[66] Defense Advanced Research Projects Agency & Information Sciences Institute, University of Southern California, "Transmission Control Protocol," Request for Comments 793, Internet Engineering Task Force, September 1981.

[67] J. Postel, ISI, "User Datagram Protocol," Request for Comments 768, Internet Engineering Task Force, August 1980.

[68] T. Ylonen, T. Kivinen, M. Saarinen, T. Rinne, S. Lehtinen, "SSH Protocol Architecture," Network Working Group, Internet-Draft, November 19, 2001.

[69] S. Bradner, A. Mankin, "IP: Next Generation (IPng) White Paper Solicitation," Request for Comments 1550, Internet Engineering Task Force, December 1993

[70] S. Deering, R. Hinden, "Internet Protocol, Version 6 (IPv6) Specification," Request for Comments 1883, Internet Engineering Task Force, December 1995.

[71] Tuecke, S., Czajkowski, K., Foster, I., Frey, J., Graham, S., Kesselman, C, Maguire T., Sandholm, T., Snelling, D., and Vanderbilt, P., Open Grid Services Infrastructure (OGSI) Version 1.0. Global Grid Forum, June 2003.

[72] "GT3 Core Key Concepts". http://www-unix.globus.org/toolkit/docs/3.2/core/key/index.html

[73] "GT4 Release Contents". http://www-unix.globus.org/toolkit/docs/development/4.0-drafts/GT4Facts/index.html#Contents

[74] "GRAM: Key Concepts". http://www-unix.globus.org/toolkit/docs/3.2/gram/key/index.html

[75] K. Czajkowski, I. Foster, N. Karonis, C. Kesselman, S. Martin, W. Smith, S. Tuecke, "A Resource Management Architecture for Metacomputing Systems", IPPS/SPDP '98 Workshop on Job Scheduling Strategies for Parallel Processing, pg. 62-82, 1998.

[76] "GT 4.0 WS GRAM Approach". http://www-unix.globus.org/toolkit/docs/development/4.0-drafts/execution/key/WS_GRAM_Approach.html

[77] GridFTP: Universal Data Transfer for the Grid. Globus Project, White Paper. http://www.globus.org/datagrid/deliverables/C2WPdraft3.pdf

[78] "GT 4.0 Component Fact Sheet: WS MDS (MDS4)". http://www-unix.globus.org/toolkit/docs/development/4.0-drafts/info/WSMDSFacts.html

[79] K. Ranganathan, I. Foster, "Simulation Studies of Computation and Data Scheduling Algorithms for Data Grids", Journal of Grid Computing, 2003, 1 (1).

[80] K. Ranganathan, I. Foster, "Decoupling Computation and Data Scheduling in Distributed Data-Intensive Applications", in 11th IEEE International Symposium on High Performance Distributed Computing. 2002. Edinburgh, Scotland: IEEE Computer Society Press.

[81] C. Dumitrescu, I. Foster, "Usage Policy-based CPU Sharing in Virtual Organizations", in 5th International Workshop in Grid Computing, 2004, Pittsburg, PA.

[82] C. Dumitrescu, I. Raicu, M. Ripeanu, I. Foster. "DiPerF: an automated DIstributed PERformance testing Framework." 5th International IEEE/ACM Workshop in Grid Computing, 2004, Pittsburg, PA.

[83] Bill Wilson. "GKrellM Monitor," http://members.dslextreme.com/users/billw/gkrellm/gkrellm.html

[84] "select(2) - Linux man page", http://www.die.net/doc/linux/man/man2/select.2.html

[85] Abhishek Chandra and David Mosberger. "Scalability of Linux Event-Dispatch Mechanisms," Proceedings of the USENIX Annual Technical Conference (USENIX 2001), Boston, MA, June 2001.

[86] N. Minar, "A Survey of the NTP protocol", MIT Media Lab, Dcemeber 1999, http://xenia.media.mit.edu/~nelson/research/ntp-survey99.

[87] T. Williams, C. Kelley. "gnuplot, An Interactive Plotting Program", http://www.gnuplot.info/docs/gnuplot.pdf

[88] The Ganglia Distributed Monitoring System: Design, Implementation, and Experience. Matthew L. Massie, Brent N. Chun, and David E. Culler. Parallel Computing, Vol. 30, Issue 7, July 2004.

[89] C. Dumitrescu, I. Foster, "GangSim: A Simulator for Grid Scheduling Studies", Cluster Computing and Grid (CCGrid), Cardiff, UK, May 2005.

[90] C. Dumitrescu, I. Foster, "GRUBER: A Grid Resource SLA Broker", GriPhyN/iVDGL Technical Report, 2005.

[91] C. Dumitrescu, I. Raicu, M. Ripeanu. "Extending a distributed usage SLA resource broker with overlay networks to support Large Dynamic Grid Environments", work in progress.

[92] I. Raicu. "Decreasing End-to-End Job Execution Times by Increasing Resource Utilization using Predictive Scheduling in the Grid", Technical Report, Grid Computing Seminar, Department of Computer Science, University of Chicago, March 2005.

[93] H. Subramanian. "Systems Performance Evaluation Methods for Distributed Systems Using Datastreams", Master Thesis, Information & Telecommunication Technology Center, University of Kansas, January 2005.

[94] C. Pistol, A. Lungu. "Deploying C++ Grid Services: Options and Performance", Technical Report, Federated Distributed Systems, Department of Computer Science, Duke University, December 2004.

[95] A. Dan, C. Dumitrescu, and M. Ripeanu, "Connecting Client Objectives with Resource Capabilities: An Essential Component for Grid Service Management Infrastructures," 2nd International Conference on Service Oriented Computing (ICSOC), November 2004, New York, NY.

[96] R. Wolski, "Dynamically Forecasting Network Performance Using the Network Weather Service", Journal of Cluster Computing, Volume 1, pp. 119-132, Jan. 1998.

[97] A. Danalis, C. Dovrolis. "ANEMOS, An Autonomous NEtwork MOnitoring System." 4th Passive and Active Measurements (PAM) Workshop 2003.

[98] U. Hofmann, I. Milouchewa1. “Distributed Measurement and Monitoring in IP Networks.“ SCI 2001/ISAS 2001 Orlando 7/2001.

[99] J. L. Hellerstein, M. M. Maccabee, W. Nathaniel Mills III, and J. J. Turek. “ETE, A Customizable Approach to Measuring End-to-End Response Times Their Components in Distributed Systems.” 19th IEEE International Conference on Distributed Computing Systems (ICDCS) 1999.

[100] C. R. Simpson Jr., G. F. Riley: “NETI@home: A Distributed Approach to Collecting End-to-End Network Performance Measurements.” PAM 2004.

[101] J. Schopf and F. Berman, “Using Stochastic Information to Predict Application Behavior on Contended Resources," Jrnl. of Foundations of CS, 2001.

[102] W. Smith, I. Foster, and V. Taylor. “Predicting Application Run Times Using Historical Information”. IPPS/SPDP 1998.

[103] P. Dinda, “A Prediction-based Real-time Scheduling Advisor”, 16th International Parallel and Distributed Processing Symposium (IPDPS 2002).

[104] D.A. Bacigalupo, S.A. Jarvis, L. He, D.P. Spooner, D.N. Dillenberger, G.R. Nudd, “An investigation into the application of different performance prediction techniques to distributed enterprise applications”, Journal of Supercomputing, 2004.

[105] S.A. Jarvis, D.P. Spooner, H.N. Lim Choi Keung, J. Cao, S. Saini, G.R. Nudd. “Performance Prediction and its use in Parallel and Distributed Computing Systems”, IEEE/ACM International Workshop on Performance Modelling, Evaluation and Optimization of Parallel and Distributed Systems, 2003.

[106] F. Vraalsen. "Performance Contracts: Predicting and Monitoring Grid Application Behavior." MS Thesis, Graduate College of UIUC, 2001.

[107] N. N. Tran. “Automatic ARIMA Time Series Modeling and Forecasting for Adaptive Input/Output Prefetching.” PhD thesis, UIUC, 2001.

[108] J. Oly and D. A. Reed, "Markov Model Prediction of I/O Request for Scientific Application," FAST 2002.

[109] D. Irwin, J. Chase, and L. Grit, “Balancing Risk and Reward in Market-Based Task Scheduling.” HPDC-13, 2004.

[110] J. Kay, and P. Lauder, “A Fair Share Scheduler,” University of Sydney and AT&T Bell Labs, 1988.

[111] "Maui Scheduler, http://www.supercluster.org/maui/." Center for HPC Cluster Resource Management and Scheduling, 2004.

[112] LSF Administrator’s Guide, Version 4.1, Platform Computing Corporation, February 2001.

[113] Condor Project, A Resource Manager for High Throughput Computing, Software Project, The University of Wisconsin, www.cs.wisc.edu/condor.

[114] OpenPBS Project, A Batching Queuing System, Software Project, Altair Grid Technologies, LLC, www.openpbs.org.

[115] I. Foster, C. Kesselman, C. Lee, R. Lindell, K. Nahrstedt, and A. Roy, "A Distributed Resource Management Architecture that Supports Advance Reservations and Co-Allocation," Proc. International Workshop on Quality of Service, 1999.

[116] “Open source metascheduling for Virtual Organizations with the Community Scheduler Framework (CSF)”, Technical White Paper, Platform, 2003.