J Grid Computing
DOI 10.1007/s10723-006-9037-5

The Design, Performance, and Use of DiPerF: An automated DIstributed PERformance evaluation Framework

Ioan Raicu · Catalin Dumitrescu · Matei Ripeanu · Ian Foster

Received: 1 September 2005 / Accepted: 25 February 2006

Abstract We present DiPerF, a DIstributed PERformance evaluation Framework, aimed at simplifying and automating performance evaluation of networked services. DiPerF coordinates a pool of machines that access a target service and collect performance measurements, aggregates these measurements, and generates performance statistics. The aggregate data collected provide information on service throughput, service response time, service 'fairness' when serving multiple clients concurrently, and on the impact of network connectivity on service performance. We have used DiPerF in various environments (PlanetLab, Grid3, TeraGrid, and a cluster) and with a large number of services. This paper provides data that demonstrates that DiPerF is accurate: The aggregate client view matches the tested service view within a few percents, and scalable: DiPerF can handle more than 10,000 clients and 100,000 transactions per second. Moreover, rapid adoption and extensive use demonstrate that the ability to automate performance characteristics extraction makes DiPerF a valuable tool.

Key words performance evaluation · Grid computing · Globus Toolkit

I. Raicu (✉) · I. Foster
Computer Science Department,
The University of Chicago,
Chicago, IL, USA
e-mail: iraicu@cs.uchicago.edu

I. Foster
e-mail: foster@cs.uchicago.edu

C. Dumitrescu
Electrical Engineering, Mathematical and Computer Science Department, Delft University of Technology,
Delft, The Netherlands
e-mail: c.dumitrescu@ewi.tudelft.nl

M. Ripeanu
Electrical and Computer Engineering Department,
The University of British Columbia,
Victoria, BC, USA
e-mail: matei@ece.ubc.ca

I. Foster
Mathematics and Computer Science Division,
Argonne National Laboratory,
Argonne, IL, USA

1. Introduction

Although performance evaluation is an 'everyday' task, testing harnesses are often built from scratch for each particular service. To address this issue, we have developed DiPerF, a DIstributed PERformance evaluation Framework, aimed to simplify and automate service performance evaluation. DiPerF coordinates a pool of machines that access a target service and client-centric performance measurements, aggregates these performance

measurements, and generates performance statistics. The aggregate data collected provide information on service throughput, service response time, service 'fairness' when serving multiple clients concurrently, and on the impact of network latency on service performance.

Actual service performance experienced by heterogeneous, geographically distributed clients with different levels of connectivity cannot be easily gauged using LAN-based testbeds. As a result, significant effort is often required to deploy and control the testing platform itself. With wide-area, heterogeneous deployment platforms like the PlanetLab [1] or Grid3 [2] testbeds, DiPerF can provide accurate estimation of the service performance as experienced by a geographically diverse set of clients. Additionally homogeneous cluster deployments allow us to conduct experiments to estimate performance in homogeneous LAN environments. The ability to conduct both LAN and WAN based evaluations of the same service makes DiPerF a useful resource planning tool for administrators managing large-scale systems, such as Grids [3].

Automated performance evaluation across a distributed testbed is complicated by multiple factors:

- *Clock synchronization* for a large set of testing machines to control performance estimation accuracy. The *accuracy* of the performance metrics collected depends heavily 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 an average synchronization error smaller than 100 ms. Additionally, DiPerF detects client failures that occur during the test and impact on reported result accuracy.
- The *heterogeneity* of WAN environments poses a challenging problem for any large scale performance evaluation platform due to different remote access methods, administrative domains, hardware architectures, and hosting environments. We have demonstrated DiPerF *flexibility* to operate in heterogeneous environments by deploying and operating it over four testbeds (Grid3 [2], PlanetLab [1], TeraGrid [33], and a cluster) with different facilities to deploy clients, manage computations, and retrieve data.
- *Coordination of a large number of resources. Scalability* of the framework itself is important in order to push a service under stress-test a service to its limits. We insure scalability by loosely coupling participating components. DiPerF has been designed and implemented to scale to at least 10,000 clients that can generate 100,000 transactions per second. In our tests DiPerF has processed up to 200,000 transactions per second via TCP and up to 15,000 transactions per second via SSH-based communication. DiPerF implementation has been carefully tuned to use the lightweight protocols and tools in order to improve scalability. For example, the communication protocol built on TCP uses a single process and the select() function to multiplex between thousands of concurrent client connections. The structures used to store and transfer the performance metrics have been optimized for space and efficiency. Similarly, each TCP connection's buffering is kept to a minimum in order to lower the memory footprint of DiPerF central node and improve scalability.

In summary, DiPerF has been designed from the ground aiming for scalability, performance, flexibility, and accuracy, and, based on the results we present in this study, we believe these goals have been achieved. After 18 months since its original implementation, DiPerF has proved to be a valuable tool for automated extraction of service performance characteristics and scalability studies and has been adopted by numerous research teams. Projects using DiPerF we are aware of include: DI-GRUBER [4, 5], predictive scheduling [7], workload performance measurements [8], GridFTP [9], WS-MDS [6, 32], and GRAM [6, 10].

The rest of this paper is organized as follows. The following section discusses related work while Section 3 presents the DiPerF design in detail and focuses on scalability issues, client code distribution, clock synchronization, client control, metric aggregation, and the performance analyzer. Section 4 covers the scalability, performance, and validation study of DiPerF while Section 5 briefly

outlines results from using DiPerF to evaluate the performance of various Globus Toolkit components. Section 6 summarizes this paper and presents future work.

2. Related Work

2.1. Distributed Performance Measurement Studies

A number of wide-area measurement projects focus on generic network monitoring and analysis, rather than on services or specifically on Grid service. We briefly summarize this work and highlight differences when compared to DiPerF.

The IETF Internet Protocol Performance Metrics (IPPM) [11, 12] working group's mission is to "develop a set of standard metrics that can be applied to the quality, performance, and reliability of Internet data delivery services." The Surveyor [13] project, grown out of the IPPM work, is focused on one-way measurements of packet delay and loss between a relatively small numbers of systems deployed world-wide. From our perspective, the most important aspect of this system is that it uses Global Positioning Satellite (GPS) clocks which enable precise clock synchronization of participating systems.

Keynote Systems, Inc. [14] owns an infrastructure of over 1,000 measurement systems deployed world wide used to evaluate the performance (e.g. observed download times) of individual Web sites and, to a lesser extent, Internet Service Providers. Keynote's system provides only DNS response time and document download time, and does not differentiate between network and server related components of response time. Keynote also publishes a 'performance index' that attempts to assign a single 'health' value to the 40 most heavily used consumer and business Web.

The CoSMoS system [15] is a performance monitoring tool for distributed programs executing on loosely coupled systems such as workstation clusters. The monitor is capable of capturing measurement data at the application, operating system, and hardware level. In order to compensate for the lack of a globally synchronous clock on loosely coupled systems, similar to DiPerF, the monitor provides a mechanism for synchronizing the measurement data from different machines.

Remos [16] provides resource information to distributed applications. Scalability, flexibility, and portability are achieved through an architecture that allows components to be positioned across the network to collect information about each local network.

Finally, Web server performance has been a topic of significant research. The Wide Area Web Measurement (WAWM) Project for example designs an infrastructure distributed across the Internet allowing simultaneous measurement of web client performance, network performance, and web server performance [17]. Banga et al. [18] measure the capacity of web servers under realistic loads. Both systems could have benefited from a generic performance evaluation framework such as DiPerF.

Each of the projects mentioned above measures or analyzes some aspect of Internet/network performance. Most focus on general network monitoring and analysis, rather than service/application performance, and the few projects that do concentrate on service level performance, have often been deployed on small testbeds and do not consider a wide-area environment; furthermore, some of these frameworks require client and/or server side source code modifications.

2.2. Grid Performance Studies

NetLogger [19] targets instrumentation of Grid middleware and applications, and attempts to control and adapt the amount of monitoring data produced. NetLogger focuses on monitoring and requires client code modification; additionally, NetLogger does not address automated client distribution and automatic data analysis.

GridBench [20] 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 characteristics of a Grid node.

The development team of the Globus Toolkit has performed extensive component testing in LAN environments [21, 22]. Some of these tests

are more complex than those we perform with DiPerF; however, they have the downside of an artificial environment, with multiple clients running on few, well connected, physical machines. We believe that our results obtained using hundreds of geographically distributed machines present a more realistic picture of performance characteristics of various Globus Toolkit components.

Gloperf [23] was developed as part of the Globus Toolkit to address the selection of hosts based on available bandwidth. 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) [24] is a distributed monitoring and forecasting system. A distributed set of performance sensors feed forecasting modules. There are two important differences compared 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. The DiPerF Framework

DiPerF [6, 10] aims to simplify and automate service performance evaluation. DiPerF coordinates a pool of machines that access a centralized or distributed target service and collect performance metrics. Centralized DiPerF components then, aggregate these performance measurements and generate performance statistics (Figure 1). The aggregate data collected provides information on service throughput, service response time, 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.

DiPerF consists of four major components (Figures 1 and 2): The *analyzer*, the *controller*, the *submitters* and the *testers*. A user of the framework provides to the controller the location of the target service to be evaluated and the client code to access the service. The controller then coordinates

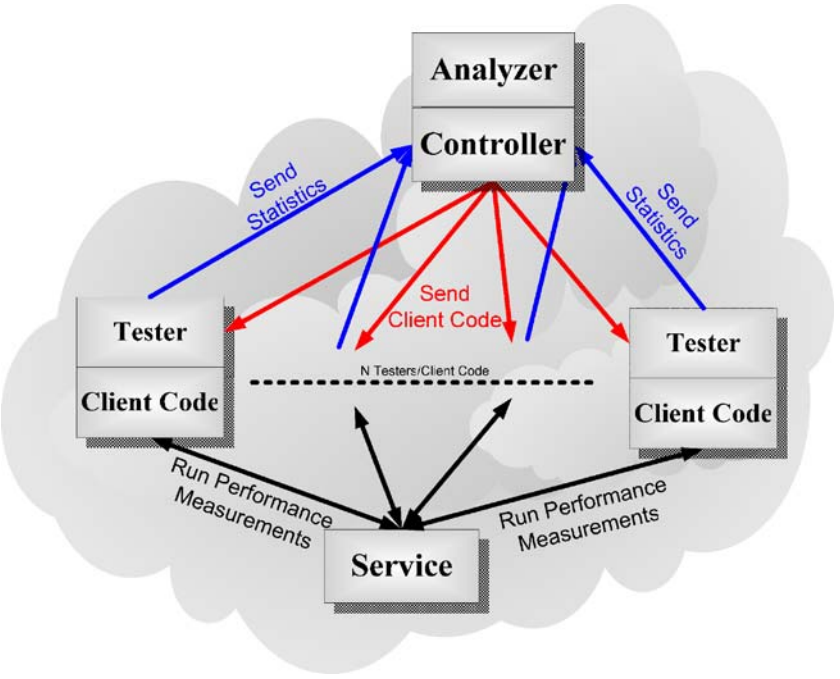

Figure 1 DiPerF framework overview.

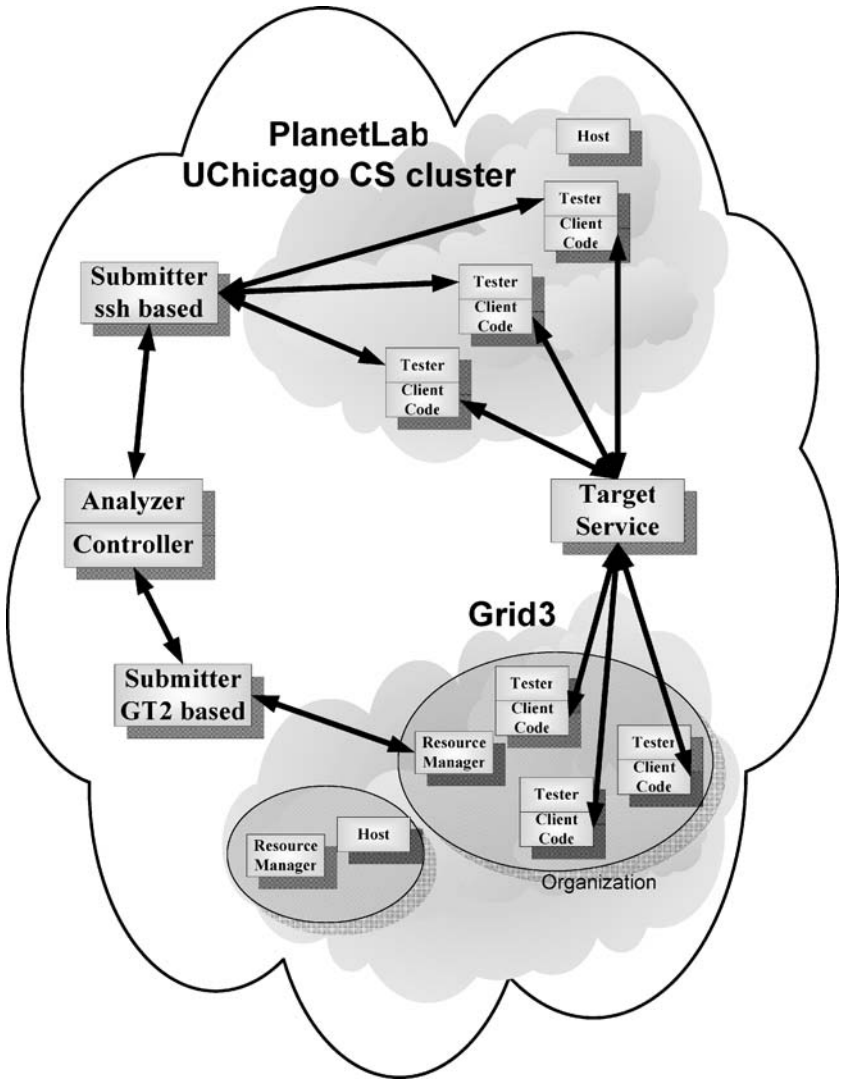

Figure 2 DiPerF deployment scenario.

the performance evaluation experiment: It distributes the client code to testers via submitters and coordinates testers' activity. Each tester runs the client code on its local machine and times their (RPC-like) access to the target service. Finally, the controller collects all the measurement data from testers and performs additional operations (e.g., reconciles time stamps from various testers) to compute aggregated performance views. Sophisticated clients can have complex interactions with the target service and return periodic feedback and user defined metrics to the tester be propagated back to the controller.

Figure 2 presents a DiPerF deployment scenario over different testbeds: PlanetLab, Grid3, and a cluster. Note the different client deployment mechanisms between the different testbeds: GlobusToolkit GRAM based submission for Grid3 and *ssh*-based tools for the other testbeds. Another difference is that in Grid3 the controller communicates only with a resource manager, and the resource manager deploys and launches the tester/client code on physical machines; in the other testbeds, the controller is directly responsible of having a complete list of available machines and communicates directly to each remote machine.

We support two different interfaces between client code and testers: First, a generic interface, for clients that are unmodified standalone

executables and make one RPC-like call to the service. And, second, a proprietary interface, for clients that pass information to the tester periodically, in a predefined format. The first approach is good in the event that the client code source code is not available or the client cannot be changed. On the other hand, the second approach is ideal if multiple service calls are made in each client execution (i.e. when just starting the client executable is expensive as is the case in many Java programs). The drawback of this latter approach is the need to instrument the client code with calls that explicitly report the necessary performance metrics. The proprietary interface includes the following five pieces of information that are streamed by clients on their standard output and parsed by testers: Metric name, value, local time, host name, and a locally unique client ID. The metric name is essential to support arbitrary metrics. The local time is necessary to be able to correctly identify the time when the sample was taken. The host name is needed to be able to identify the origin host of the metric sample. Finally, the locally unique client ID allows for multiple concurrent clients on the same physical machine.

The framework is supplied with a set of candidate nodes for client placement, and selects those available as testers. We plan to extend the framework to select a subset of available tester nodes to satisfy user specified requirements in terms of link bandwidth, latency, compute power, available memory, and/or processor load. In the 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 throughput, size of data transmitted, time to complete a subtask), measured by clients can be optionally reported to the testers and eventually back to controller for statistical analysis. As testers send performance data to the controller while the test is progressing service evolution and performance can be visualized 'on-line.'

Finally, we address failures by making it the clients' responsibility to report them. From DiPerF's point of view, errors are simply just a special performance metric. A client can fail because of various reasons: Predefined timeout enforced by the tester, client machine problems (e.g., the client fails to start due to insufficient resources), or service machine related (e.g., service denied or service not found).

3.1. Performance and Scalability Issues

We have made a number of improvements to our initial implementation of DiPerF [10] to improve scalability and performance. Using a submitter and testers built on top of end-user tools such as *ssh*, DiPerF was limited to handling only about 1,000 clients. We therefore concentrated on reducing the amount of processing per performance reporting transaction, the memory footprint of the controller, and the number of processes being spawned throughout the experiment.

In order to make DiPerF as flexible as possible for a wide range of configurations, we have stripped down the controller from most of its online data processing tasks and moved them to the offline data analysis component. This improved controller scalability, freeing the CPU of unnecessary load throughout an experiment. A more complex controller version is also made available, if results need to be viewed in real-time as the experiment progresses. 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 for multiple testers on the same node by identifying a tester by the node name followed by its process ID.

To alleviate the biggest bottleneck we have identified, namely communication based on *ssh* tools, we have implemented two other pairs of submitters and testers on top of TCP and UDP. Running TCP we can also control connection buffer sizes to reduce memory usage, especially since we can sacrifice buffer size without affecting overall system performance due to the low per-connection communication volume.

Finally, in order to achieve the best performance with the implementation of the communication over TCP or UDP, we use a single

controller process which uses the select() system call [25] to provide synchronous I/O multiplexing between 1,000s of concurrent connections. select() waits for a number of file descriptors (found in the fd_set variable) to change status based on a specified timeout. 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 1,024. This means that any given select() function can only have 1,024 file descriptors (i.e. TCP sockets) that it is listening on. This is a serious limitation, and would limit any single process DiPerF implementation over TCP to only 1,024 clients. After modifying some system files (required root access) to raise the constant size fd_set from 1,024 to 65,536, we were able to break the 1,024 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 1,024, this did not seem to be a problem, but with an fd_set size of 65,536, it quickly became a bottleneck. The solution 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. We are currently porting the select() based implementation to a /dev/poll-based implementation[26] that is should provide even lower overheads and improve the performance when using a large number of TCP connections.

With these 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 increased from 1,000 to 230,000 performance reporting transactions per second depending on the number concurrent clients and the communication protocol used.

3.2. 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 tools are deployed on just about any Linux/Unix, we base our distribution system on scp-like tools wherever possible. More specifically, to deploy client code, DiPerF uses rsync in a Unix-like environment (e.g., PlanetLab, clusters) and GT2 GRAM job submission mechanisms in a Grid environment (e.g., Grid3).

3.3. Clock Synchronization

DiPerF relies on time synchronization when aggregating results at the controller; therefore, automatic time synchronization or clock reconciliation between participating client nodes need to be provided. In the latter case synchronization is not be performed on-line; instead, the controller computes the offset between local and global time and applies that offset when analyzing aggregated metrics. The solution to possible clock drift that might affect result accuracy is to compute clock offsets during an experiment 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 (e.g., NTP [27]). In a previous large scale study [27] geographically distributed NTP synchronized hosts had a mean delay of 33 ms, median 32 ms, and a standard deviation 115 ms from their peer hosts. At the time of our experiments, although PlanetLab testbed used NTP to synchronize clocks we have found nodes with synchronization differences in the thousands of seconds. As a result, DiPerF implements its own clock reconciliation technique at the user level as a backup mechanism.

DiPerF handles time reconciliation using a centralized time-stamp server that allows time mapping to a common base. The time-stamp server is lightweight and can easily handle thousands of concurrent clients. In order to reduce the effects of clock drift, each client regularly acquires a new time-stamp. As the time server is lightweight and time synchronizations are relatively rare during an experiment (every 300 s), we estimate that the time server can handle more than a million concurrent clients. A second important concern is the case of Grid like environments where often a tester can run on a cluster node that with no direct access to the outside Internet. In these situations, the testers synchronize by means of light-weight time proxies running on the cluster head-nodes.

We have measured the latency from over 100 PlanetLab nodes to our timestamp server deployed at the University Chicago over a period of

two h. During this interval the (per-node) latency in the network remained fairly constant and the majority of the clients had a network latency of less than 80 ms. 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 synchronization error can be at most the one-way latency. Using our custom synchronization component, we observed a mean of 62 ms, median 57 ms, and a standard deviation 52 ms for the time skew between nodes in our PlanetLab testbed. Given that the response time of the relatively heavy weight services that have been evaluated (e.g., GRAM, GridFTP, MDS) in this paper is at least one order of magnitude larger, we believe the clock synchronization technique implemented does not distort the results presented.

3.4. Client Control and Performance Metric Aggregation

A DiPerF experiment works as follows: The controller starts tester incrementally with a predefined delay specified in a configuration file in order to gradually build up the load on the target service and sends test descriptions. The most important description parameters are: The duration of the test experiment, the time interval between client invocations (think time), an upper bound for client processing time to detect failed calls due to unresponsive services, 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 invoke clients, collect service response times and other service metrics and report them back to the controller. The controller's job then 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 timestamp, it becomes feasible 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 (Figure 3). The data analysis process is completely automated (including graph generation) at the user-specified time granularity.

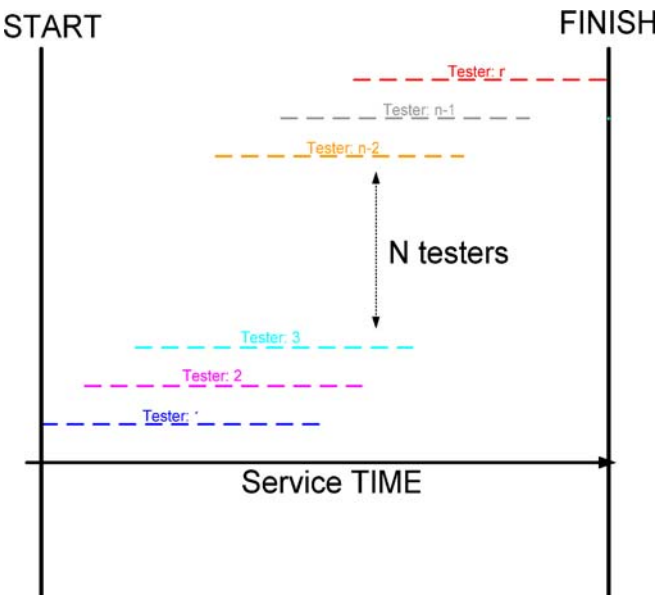

Figure 3 Aggregate view at the controller. Each tester synchronizes its clock with the time server every 300 s.

3.5. Performance Metrics Collected and the Performance Analyzer

While using DiPerF, we have realized the need for a specialized component to analyze performance data automatically for end-user. Thus, we introduced the performance analyzer, which has been implemented in C++ and currently consists of over 4,000 lines of code. Its current implementation this is an off-line process: It assumes that all performance data is available for processing; in the future, we plan to port the current performance analyzer to support on-line analysis of incoming performance data.

We have focused on flexibility in handling large data analysis tasks completely unsupervised. The performance analyzer is 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 can accumulate more than one million performance samples over a period of and hour, but after the performance analyzer summarizes the data for one sample per second, the end result is reduced to 3,600 samples.

We also introduce the *performance metrics* considered by DiPerF's analyzer. While the performance metrics of interest to the user may vary from case to case, and our system allows the introduction and processing of user specified metrics, we use the following minimal set of preconfigured metrics for all the experiment we report in this paper:

- *service processing time*: The time from when a client issues a request to the moment a reply is received minus the roundtrip time to the service and minus the execution start-up time of the client code. This metric is measured from the point of view of the client.
- *service throughput*: Number of requests completed successfully by the service averaged over a short time interval specified by the user (e.g., a second or a minute) in order to reduce the large number of samples. To make the results easier to understand most of the graphs below also present moving averages.
- *offered load*: Number of concurrent service requests (per second).
- *service utilization* (per client): Ratio between the number of requests completed for a specific client and the total number of requests completed by the whole service during the time the client was active.
- *service fairness*: The standard deviation in service utilization measured when all clients are active concurrently.
- *job success rate* (per client): The ratio of jobs that were successfully completed for a particular client.
- *job fail rate* (per client): The ratio of jobs that failed for a particular client.

The performance analyzer has a few additional features. First of all, it has an option to verify that the raw input data conforms to the correct formatting requirements, and fixes (mostly by deleting) any inconsistencies it finds. Second, for all of the above metrics that are computed per client they 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 under stress. The output resulting from the performance analyzer can be automatically graphed using gnuplot [28] 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 paper.

We believe that the end product from the Performance Analyzer can be used to create performance models for the tested services to estimate service performance as a function of service load.

4. DiPerF Scalability and Validation Study

This section presents a performance and scalability evaluation of DiPerF main components individually (the controller, the time server and the data analyzer) then an evaluation of the entire framework testing a lightweight service that that requires DiPerF to generating and handling a high test load. The protocols and the various components are depicted in Figure 4. Note that we have used DiPerF as the primary tool to coordinate the scalability study of each component and of DiPerF itself.

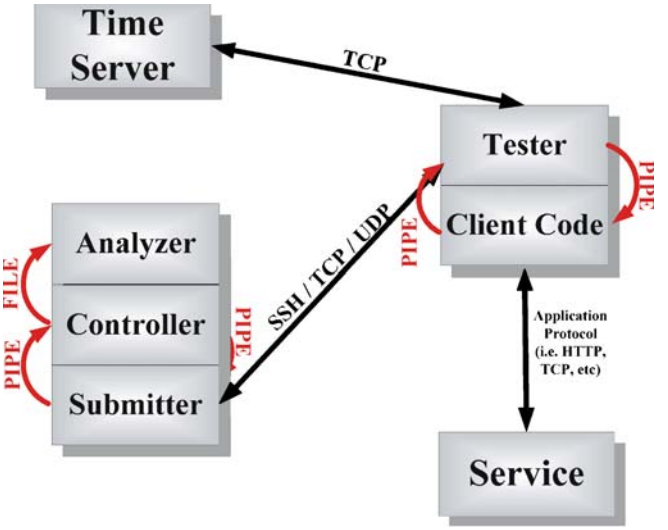

Figure 4 Components and communication overview.

4.1. Controller Performance

Figure 5 presents the performance and scalability of the controller using the three basic communication methods implemented: TCP, UDP and *ssh*. We are mainly interested in two performance metrics: The number of concurrent testers the controller can handle and the rate of performance reporting transactions that can be handled.

Note that TCP offers the best performance for almost all cases except for very large number of concurrent clients (more than 30,000). At peak, the controller using TCP can process over 230,000 transactions per second, while at the lowest point, with 60,000 concurrent clients, it could still process over 15,000 transactions per second. It is interesting that we were able to reach 60,000 clients especially that TCP port numbers are 8 bit values, which limits the number of concurrent TCP connections to 65,000. UDP offered performance is somewhat lower than with TCP, but it is much more consistent irrespective of the number of concurrent clients. Overall, UDP achieved about 80,000 transactions per second up to the 80,000 concurrent clients we tested. Because the UDP implementation did not provide a reliable communication protocol, processing that had to be done per transaction increased in order to keep

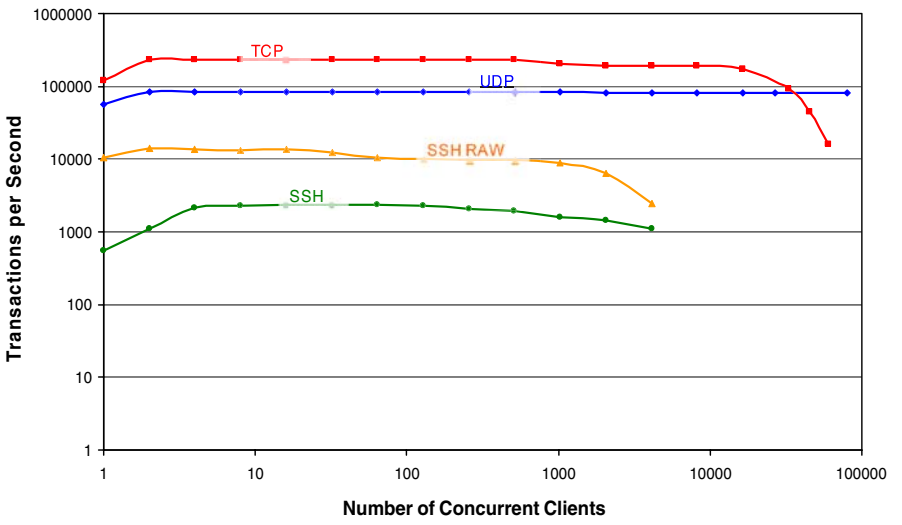

Figure 5 Summary of communication performance and scalability. The figure presents the controller throughput as a function of the number of concurrent clients handled for various protocols: UDP, TCP, SSH RAW, and SSH. Note the log scales on both axes.

track of the number of messages lost and the number of concurrent machines.

For *ssh*, we have investigated two alternatives: *ssh*-raw which simply wrote all transactions to a file for later analysis, while *ssh* performed some basic analysis in real-time as it collected all the metrics, and kept all metrics in memory for later faster analysis. When using *ssh* the controller was able to achieve 15,000 transactions per second without any analysis in real time and about 2,000 transactions per second with some basic analysis. Finally, in Figure 5, note the increase in performance from one to two concurrent clients across all communication protocols. This owes to the fact that a dual processor machine was used in handling the multiple client requests; therefore, at least two concurrent clients were needed to reach the full capacity on both processors.

4.2. Time Server Performance

Time synchronization is an important component of DiPerF, and hence the performance of the time server is determinant for overall DiPerF performance. The time server uses TCP as its main communication protocol to reply to time queries. It is interesting to note in Figure 6 that the service response time remained relatively constant (a little over 200 ms) throughout the entire experiment.

A total of 1,000 clients were used for a total peak rate of about 2,000 queries per second. Note that the network roundtrip time (marked network latency in Figure 6) is nearly half the service response time; the difference (about 100 ms) can be attributed to the time it takes a packet to traverse the network protocol stack, instantiate a TCP connection via a three-way handshake, transfers a small amount of information, and tears down the TCP connection. Our implementation is based on a simple TCP server that accepts incoming connections to request the 'global' time. We are currently working on implementing a proprietary client/server model based on the UDP/IP protocol in order to make the time server even more light weight, more scalable, and reduce the response times the clients observe in half by avoiding the three-way handshake that our current implementation based on TCP has. In our new UDP/IP based implementation, we will have to address reliability issues by adding some simple reliability

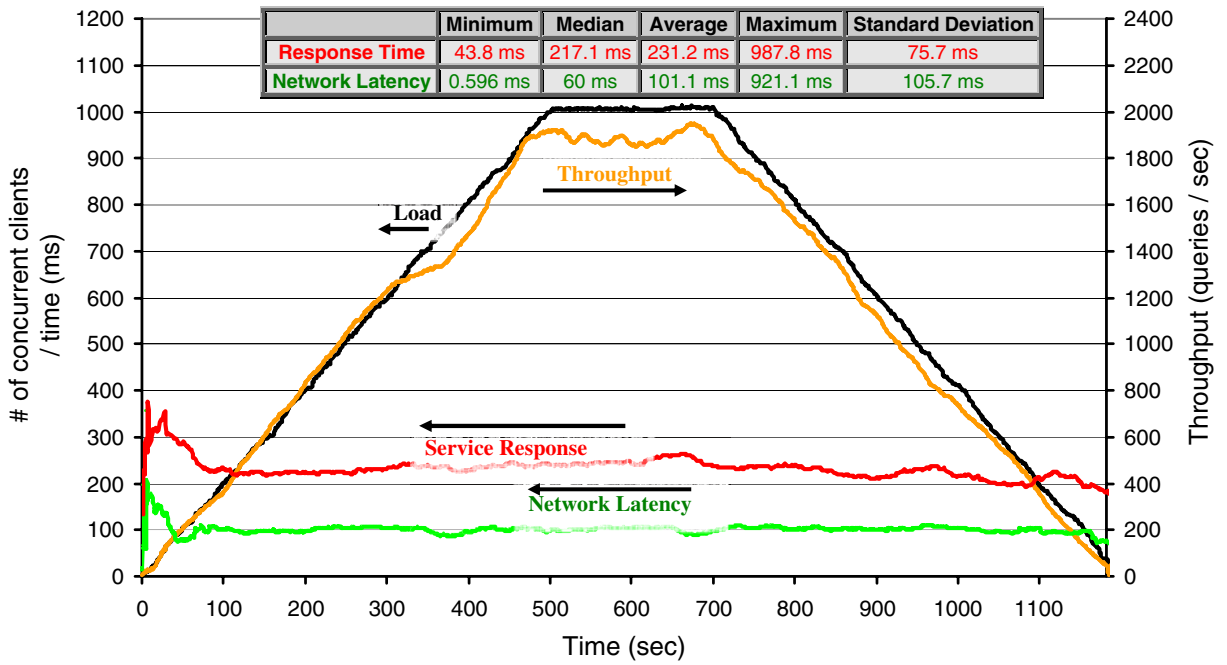

Figure 6 Time server performance with 1,000 clients. *Arrows* indicate the *y*-axis (*left* or *right*) associated with each particular metric.

mechanisms, such as retransmissions and acknowledgements. Figure 6 shows the performance of the time server as it scales up to 1,000 concurrent clients; the server exhibits a consistent throughput and response time with a large number of concurrent clients.

4.3. Analyzer Performance

The analyzer component is currently implemented as an offline process, and hence its performance is not crucial. Because of the many variables that affect the performance of the analyzer, it is hard to generalize the results. Table 1 summarizes analyzer performance for four different runs.

All four runs first verified that transaction file that it contained no errors, after which it performed its analysis to extract three metrics: Load, throughput, and response time. The main deciding factor in the performance of the analyzer is the number of machines and the number of transaction that need to be processed. For example, for a relatively small run with 40 clients, nearly 3,000 transactions which spanned 1,000 s, it took only 1.5 s to both verify that data and extract the three metrics. On the other hand, for a larger run, which involved 3,600 clients and 354,000 transactions, the total time grew significantly to almost 6 min. The observed transactions per second throughput achieved by the analyzer varied from 1,000 to as high as 9,000 transactions per second.

4.4. Overall DiPerF Performance

In the previous few sections, we have presented the performance of the various individual components. This section explores the overall DiPerF performance. We test the scalability limits of DiPerF, with the simplest service possible: A services that does not require any input and echoes back a single character. This experiment was carried out over the 20 node cluster connected by either 100 Mbps network links. Each client was configured to transmit one transaction per second over the TCP-based communication protocol. In Figure 7 note the near perfect performance up to about 45,000 clients, when every client request is correctly handled. Above 45,000 concurrent clients, throughput started dropping to about 15 K transactions per second. Note that due to limitations of TCP's port number (an 8 bit value), regardless of how powerful the hardware is, this implementation cannot support more than 64,000 concurrent clients. When the controller is configured to use UDP based protocols, it is able to handle up to 80,000 transactions per second from 80,000 concurrent clients generating one transaction per second each.

Performance when using SSH-based communication is clearly lower. The main limitation results from the fact that each remote client instantiates a separate SSH connection, which runs in an isolated process. Therefore, for each new client, there is a new process starting at the controller. Additionally, each process has a certain memory footprint, and each TCP connection has the send and receive buffers set to the default value (in our case, 85 KB) and memory usage can quickly become a bottleneck. In running this experiment with 4,000 clients, DiPerF used the entire 1 GB of RAM and the entire 1 GB of swap allocated. When DiPerF operates solely out of memory, it is

Table 1 Analyzer performance.

Number of machines	Test length	Number of transactions	Memory footprint (MB)	Time quantum (s)	Time (s)	Time/trans (ms)	Trans (s)
40	1,000	2,900	0.54	1	1.6	0.6	1,812.5
200	11,000	45,000	26	1	5.1	0.1	8,840.9
1,700	6,500	671,000	146	1	166.5	0.2	4,030.0
3,600	12,000	354,000	505	1	359.0	1.0	986.1

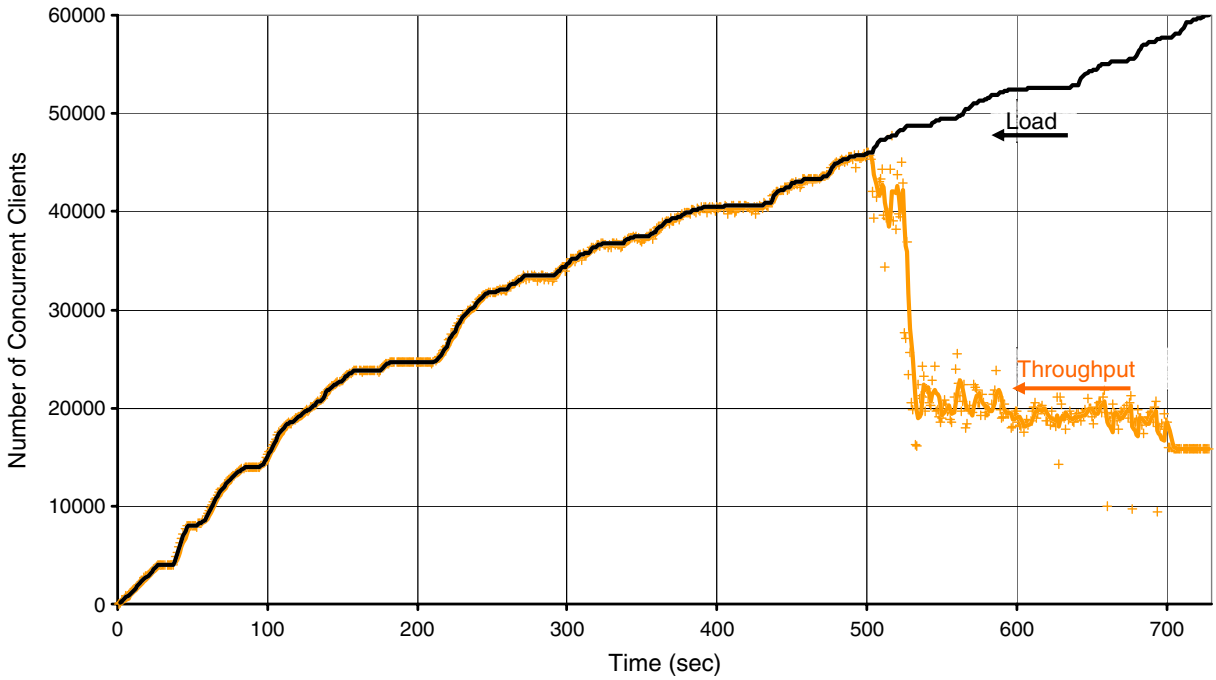

Figure 7 Echo performance via TCP-based communication protocol.

able to achieve over 2,000 transactions per second, while at the peak with 4,000 concurrent clients, it can only achieve about 1,000 transactions per second.

4.5. Validation

This section validates the results produced by DiPerF: We test a GridFTP service and compare the aggregate client view generated by DiPerF and the server view produced by the Ganglia Monitor [29]. The Ganglia Monitor reports server performance once every 60 s (230 samples for the duration of our experiment) a much lower granularity than DiPerF which generates about 340 performance reporting transactions for each Ganglia sample. As a result Ganglia reported performance appears 'smoother' in Figure 8 especially for fast changing metrics. For example, the load metric changed relatively slow, and was monotonically increasing, which produced about 1% of difference between DiPerF and Ganglia. On the other hand, the throughput observed a higher difference of about 5%. We believe this is due to different sampling resolutions between DiPerF and Ganglia.

Figure 9 presents a scenario where both the server and clients generate samples once a second. DiPerF generates nearly 2,000,000 transactions for each metric to produce client views for the load and for the throughput depicted in Figure 9. The server monitor reported the server performance once every second; due to there being 1,000 concurrent clients, this still generated nearly 1,000 client view samples for every server view sample. Due to a finer granularity of samples from the server side, the results are better than those in Figure 8; for the load, we obtained less than 1% difference on average, while for the throughput, we obtained about 3% difference on average. Nevertheless, Figures 8 and 9 show that large scale distributed testing aggregate client view can match the server's central view quite well with only a few percent metric value discrepancies on average.

5. The Utility of DiPerF: Empirical Case Studies

After only one year since its original implementation, DiPerF has proved to be a valuable tool

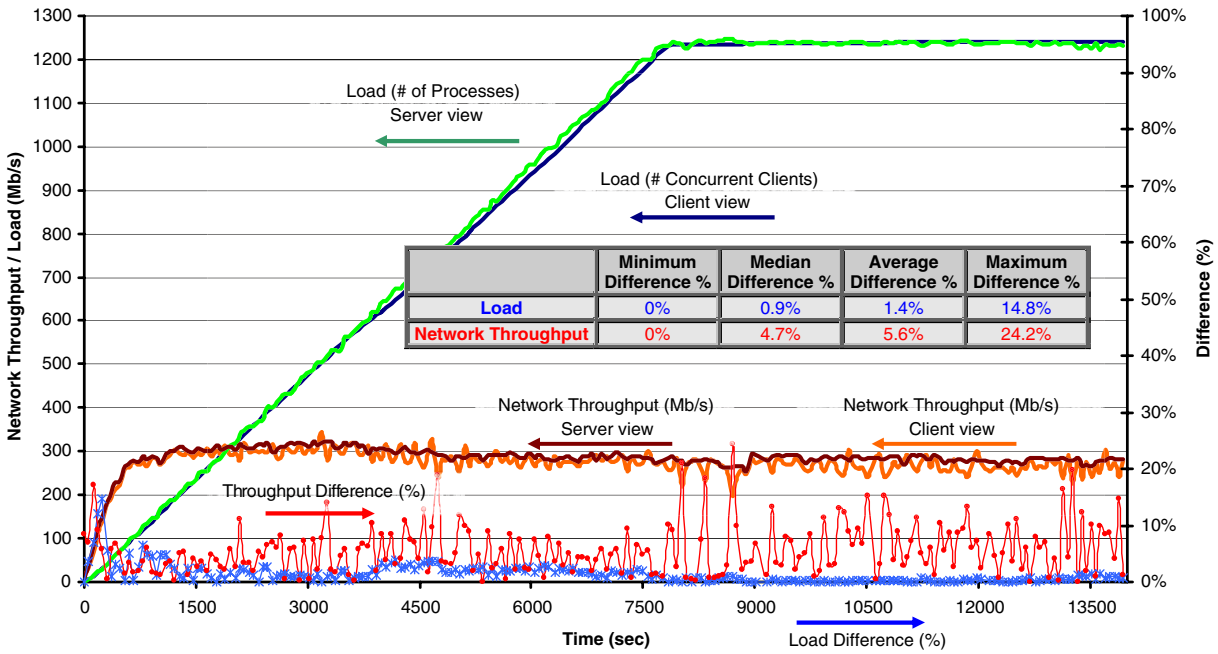

Figure 8 DiPerF validation against the Ganglia monitor using GridFTP and 1,300 clients in PlanetLab. *Arrows* indicate the *y*-axis corresponding to each metric.

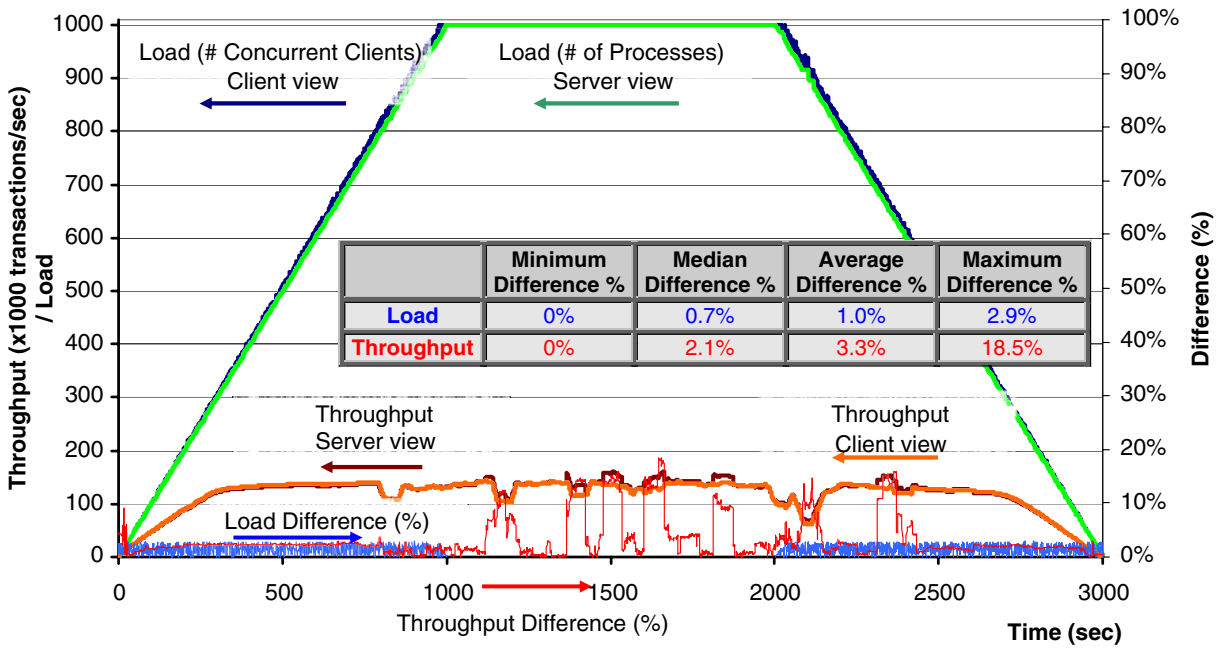

Figure 9 DiPerF Validation against a TCP server using a TCP client and 1,000 clients in PlanetLab.

for automated performance characterization and scalability studies and has been adopted by a number of research teams. Some of the projects using DiPerF, including some of our own work, we are aware of are: DI-GRUBER [4–6], Predictive Scheduling [7], workloads performance measurements of [8], GridFTP [6, 9], WS-MDS [6, 32], and GRAM4 [6, 10]. This section presents a few examples of DiPerF usage and some of the interesting findings using to the DiPerF tool. We cover a small subset of performance evaluation of three important Globus Toolkit components: The file transfer service (GridFTP), job submission (GRAM), and information services (WS-MDS). Before we delve into the experimental, we briefly discuss the testbeds used in these experiments.

5.1. Testbeds

This section covers the five testbeds (Grid3, PlanetLab, TeraGrid, and two clusters at The University of Chicago) that we have used for experiment. 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 as they have different characteristics:

- The *Grid3* collaboration has deployed an international Data Grid with dozens of sites and thousands of processors. The facility is operated jointly by the U.S. Grid projects iVDGL, GriPhyN and PPDG, and the U.S. participants in the LHC experiments ATLAS and CMS. Participation is combined across more than 25 sites which collectively provide more than 4,000 CPUs and over 2 TB of aggregate memory. The resources are used by seven different scientific applications, including three high-energy physics simulations and four data analyses in high energy physics, bio-chemistry, astrophysics and astronomy. More than 100 individuals are currently registered with access to the Grid, with a peak throughput of 500–900 jobs running concurrently [30].
- *PlanetLab* [31] 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 third-party network-wide services that define PlanetLab, a principle referred to as unbundled management. PlanetLab is deployed on over 500 nodes (Linux-based PCs) distributed over more than 150 locations around the world. Due to the large geographic distribution observed latencies and achieved bandwidth vary greatly across the testbed.
- *The TeraGrid* (TG) [33] is an open scientific discovery infrastructure combining leadership class resources at eight partner sites to create an integrated, persistent computational resource. The deployment of TeraGrid brings over 40 TFLOPS of computing power (tens of thousands of compute nodes) and nearly 2 PB of storage, and specialized data analysis and visualization resources into production, interconnected at 10–30 GBps via a dedicated national network. The initial prototype will be deployed at The University of Chicago / Argonne National Laboratory (UC/ANL) site, which has 96 IA-32 nodes and 62 IA-64 nodes as part of the TG. Each IA-32 node has dual 2.4 GHz Xeon processors, 4 GB RAM, and SuSE v8.1 Linux OS; each of the IA-64 nodes has dual 1.5 GHz Itanium 2 processors, 4 GB RAM, and SuSE v8.1 Linux OS. The nodes are all connected via 1 Gb/s Ethernet network within the site.
- *The University of Chicago CS cluster* (dubbed 'UC cluster' in the rest of this paper) contains over 100 machines that are remotely accessible. The majorities of these machines are running Debian Linux 3.0, have AMD Athlon XP Processors at 2.1 GHz, have 512 MB of RAM, and are connected via a 100 Mb/s Fast Ethernet switched network. Furthermore, all machines share a common file system via NFS (Network File System).

5.2. GridFTP Performance Study

In order to complement the LAN-based GridFTP server performance study [9] we have performed an additional evaluation in a WAN environment mainly targeting the scalability of the GridFTP server. The additional test cases can also be found in more detail in previous work [6].

The metrics collected by DiPerF's clients, defined in Section 3.5, are: Service processing time, service throughput, and offered load. Additionally, Ganglia monitoring suite is installed at the server and monitors the number of processes at server node, CPU utilization and memory used at the server, the network usage, computed from the volume of traffic flowing into the server since we were performing uploads from clients to the server.

For each set of tests, the caption below each figure will describe the particular configuration of the controller which yielded the respective results. In the figures below, each series of points represents a particular metric and is also approximated using a moving average over a 60 point interval, where each graphs consists of anywhere from 1,000s to 100,000s of data points. For all tests the testers synchronize their time every 5 min against a time server running on UC cluster node.

5.2.1. *GridFTP Scalability*

Figure 10 shows results obtained with 1,800 clients mapped in a round robin fashion on 100 Planet Lab hosts and 30 UC cluster nodes targeting a GridFTP server located in Los Angeles running on a two-processor 1,125 MHz Intel machine with 1.5 GB memory, 1 GBps Ethernet network connection, and Linux 2.6.8.1 OS with Web100 patches. A new client is created once a second. Each client runs for 2,400 s and during this time repeatedly requests the transfer of a 10 MB file from the server's disk to the client's /dev/null. A total of 150.7 GB are transferred in 15,428 transfers. The left axis indicates load (number of concurrent clients), response time, and memory allocated, while the right axis denotes both throughput and server CPU utilization. The dots in the figure represent individual client response times, while each of the lines represents a 60-s running average.

The server sustained 1,800 concurrent requests with just 70% CPU load and 0.94 MB memory per request. Furthermore, CPU usage, throughput, and response time remain reasonable even when paging is occurring. Total throughput reaches 25 MBps with less than 100 clients and exceeds 40 MBps with around 600 clients. It is interesting to see that the throughput reached around 45 MBps (or 360 MBps) and stayed consistent throughout despite the fact that the server ran out of physical memory and started swapping memory out; with the OS footprint in the neighborhood of 100 MB, and the 1,700 MB of RAM used by the GridFTP server to serve the 1,800 concurrent clients, the system ended up using about 300 MB of swap and the entire 1.5 GB of RAM. Although the CPU utilization was significant, but with 25% CPU resources available and plenty more swap space left, the server appears to be able to handle additional clients. From a CPU point of view, we are not sure how many more clients it could support since, as long as the aggregate server throughput does not increase, it is likely that the CPU will not get utilized significantly more. Another issue at this scale of tests is the fact that most operating systems have hard file descriptor usage limits. With 1,800 clients, the experiment saturated the network link into the server, but the server's raw resources (CPU, memory) were not saturated.

When comparing some of the metrics from Ganglia and those of DiPerF, it is interesting to note that the memory used follows pretty tight the number of concurrent clients. In fact, we computed that the server requires about 0.94 MB of memory for each new client it has to maintain state for; we found this to be true for all the tests we performed within ±1%. Another interesting observation is that the CPU utilization closely mimics the achieved throughput, and not necessarily the number of concurrent clients. This clean separation makes it relatively easy to plan capacity (hardware characteristics for the deployment platform) in order to achieve the desired quality of service provided by the server.

5.2.2. *GridFTP Fairness*

In order to quantify the fairness of the GridFTP server, we investigated the fairness at two levels.

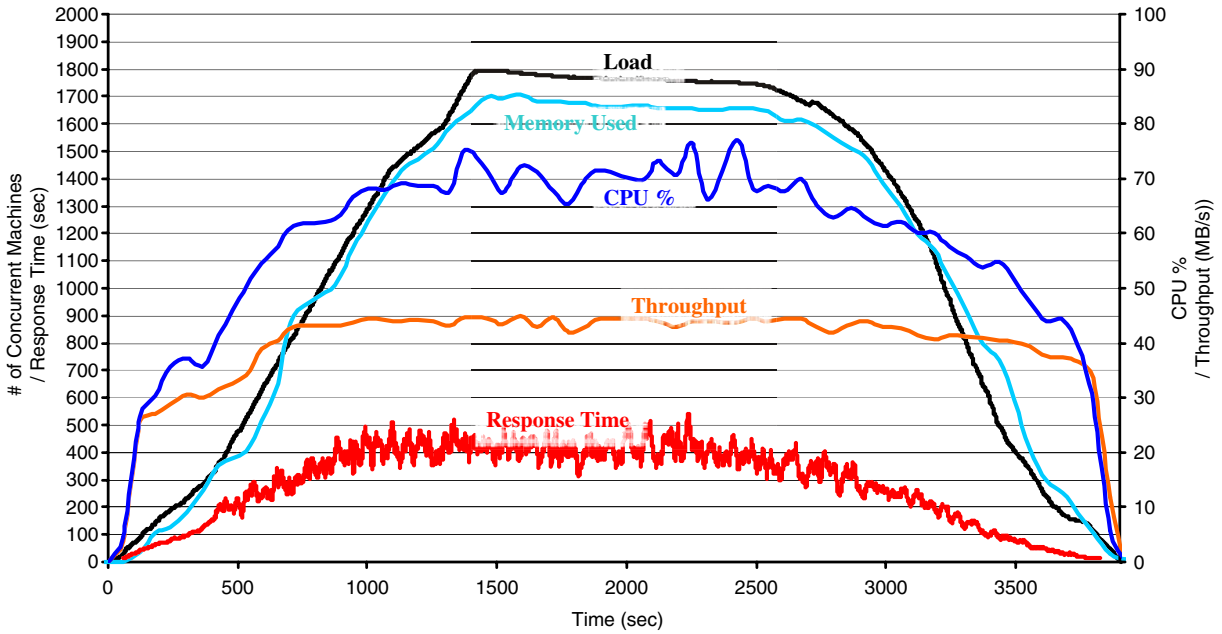

Figure 10 GridFTP server performance with 1,800 clients running on 100 physical nodes in PlanetLab and 30 UC cluster nodes. Tunable parameters used: 1,800 concurrent clients, starts a new client every 1 s, each client runs for 2,400 s; 150.7 GB of data transferred over 15,428 file transfers; *left axis* – load, response time, memory; *right axis* – throughput, CPU percent.

First, we look at fairness between the different machines (about 100 nodes in PlanetLab located in the USA) in relation to their network connectivity to ISI where the server is located. Second, we look at the fairness among the different clients running on the same physical pair of nodes. The results presented in this sub-section are the subset of the entire experiment during which all clients were running in parallel; in other words, we omitted the data when the number of clients were increasing and decreasing.

Figure 11 shows the amount of data transmitted from each machine (note that each machine ran 15 clients each). The *x*-axis represents the network latency to ISI, and the *y*-axis represents available bandwidth to ISI server as measured using *iperf* available bandwidth measurement tool. The size of the bubble represents the amount of data transmitted by each machine throughout the 3 h experiment; the largest bubbles represent a transfer amount of over 7 GB, while the smallest bubble represents a transfer of just 0.5 GB. As expected, latency plays a significant role in determining achieved TCP throughput and essentially in the achieved performance of the GridFTP client. Based on the results Figure 11, we observe that GridFTP server generally gives a fair share of resources to each machine according to available bandwidth. However, a small number of machines transfer a significantly smaller amount of data than indicated by the available bandwidth.

Figure 12 captures fairness between various competing flows running between the same pair of nodes (i.e., the same client node and the ISI server). The machines have been ordered according to their connectivity, with the best connected machines first. The (blue) horizontal line represents the average data transferred per client of a particular machine, while the vertical error bars represent the standard deviation for traffic generated by all clients running on the same machine. Note that the standard deviation is small, with only a few percent of deviation of performance from client to client running on the same host. Overall, the GridFTP server seems to allocate resources fairly across large number of clients (1,000 s), and across a heterogeneous WAN environment.

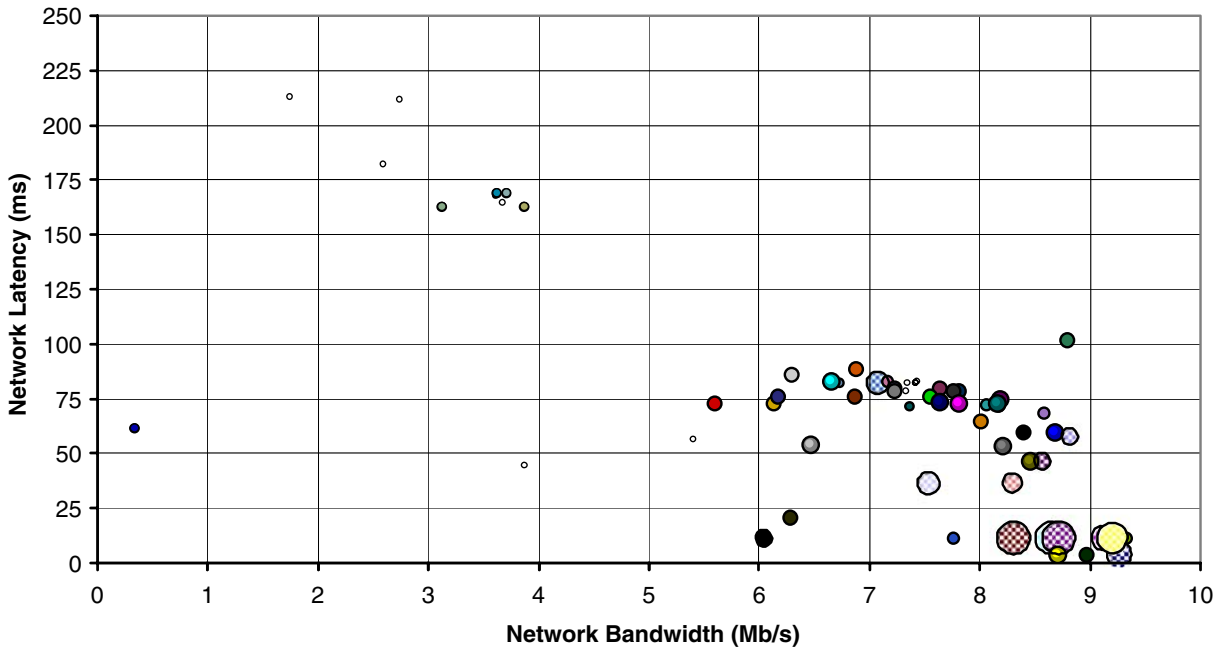

Figure 11 GridFTP generated traffic per machine.

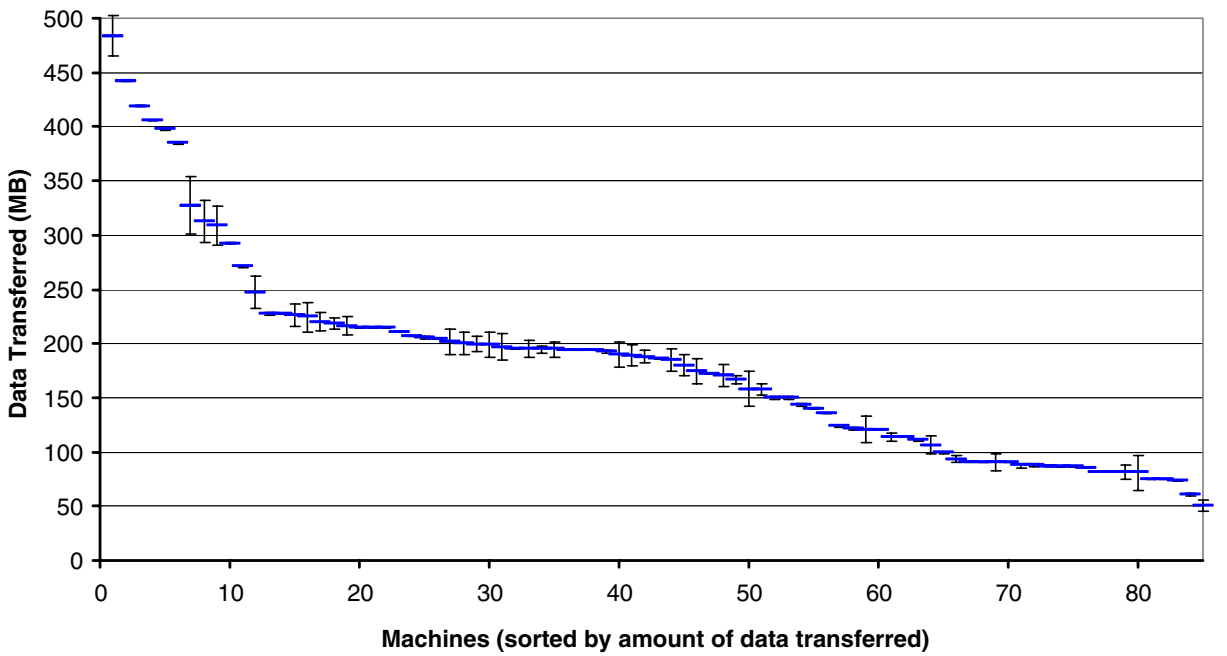

Figure 12 GridFTP resource usage per client using 10 MB files.

5.3. GT3.9.4 WS-GRAM Performance Study

We have evaluated the job submission service (GRAM) bundled with the Globus Toolkit 3.9.4 [10]. We ran the service on a AMD K7 2.16 GHz machine with 2 GB of memory located at The University of Chicago and about 115 client machines distributed over the PlanetLab testbed for clients.

As in the previous section the metrics collected by DiPerF are: Service processing time, service throughput and offered load. In the figures below, each series of points presents a particular metric and is also approximated by a moving average over a 60 point interval.

Figure 13 presents the performance of WS-GRAM C clients accessing the Java-based WS-GRAM service (version 3.9.4). We note the dramatic performance improvements compared to the previous service implementation shipped with Globus Toolkit 3.2 we measured in previous work [10]. We observe both better scalability (from 20 to 69 concurrent clients) and performance (from 10 jobs/min to over 60 jobs/min). We also note that response time steadily increase with the increased number of clients. The throughput increase levels off over about 15∼20 clients, which indicates that the service is saturated and adding increasing offered load only increases response time as requests are queued at the service.

5.4. WS-MDS Index Scalability and Performance Study

We have also evaluated the performance of the metadata and directory service (WS-MDS) bundled with the Globus Toolkit versions 3.9.5 and 4.0.1. in local- and wide-area network settings. As in the previous section the metrics collected by DiPerF are service processing time, service throughput, and offered load.

5.4.1. *WS-MDS Performance in a Wide Area Network Setting*

Figure 14 presents an experiment in which we have used a dual AMD Opteron machine with 1 GB of RAM and 1 GBps NIC located at the University of Chicago to host the WS-MSD server and, as clients, 97 PlanetLab machines and three other machines located in the same local network as the server. The main goal this experiment was to

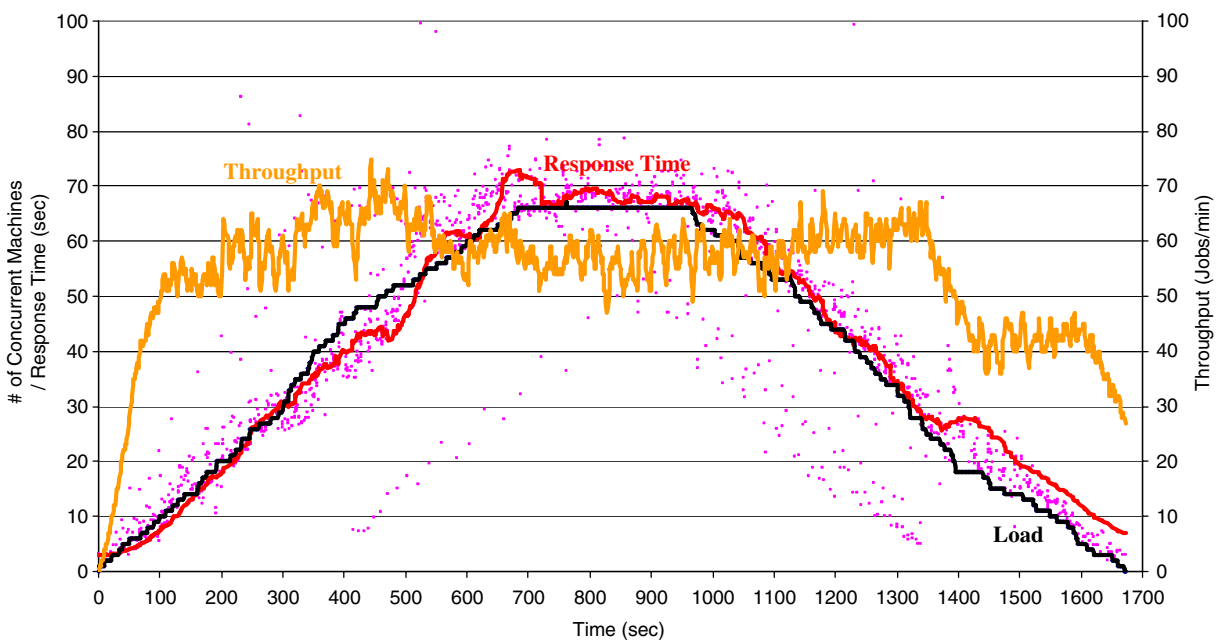

Figure 13 Globus Toolkit 3.9.4 WS GRAM client (C implementation) and service (JAVA implementation) performance. Tunable parameters: 69 concurrent client nodes, a new client every starts 10 s, each client runs for 1,000 s.

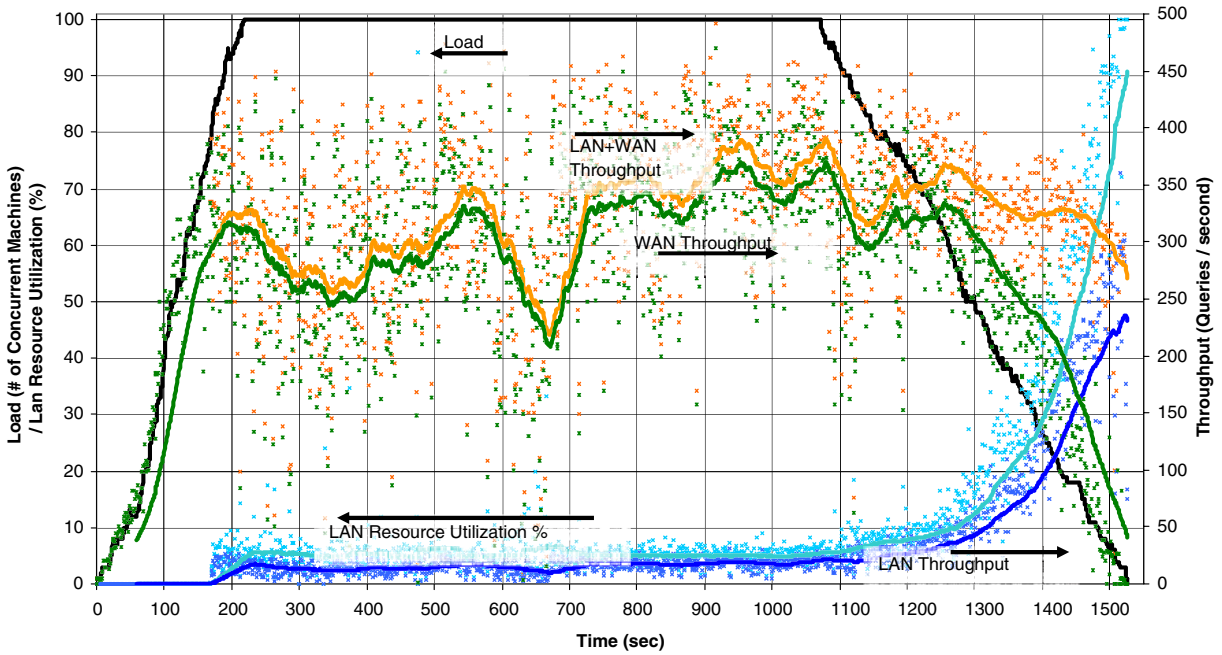

Figure 14 WS-MDS Index with 97 PlanetLab clients and three other clients hosted in the same LAN. PlanetLab clients are started (and stopped) first. Arrows indicate the *y*-axis corresponding to each metric.

explore service fairness: How much the remote clients (hosted on the PlanetLab nodes) contribute towards the overall achieved throughput. As a reminder, the remote clients are connected via

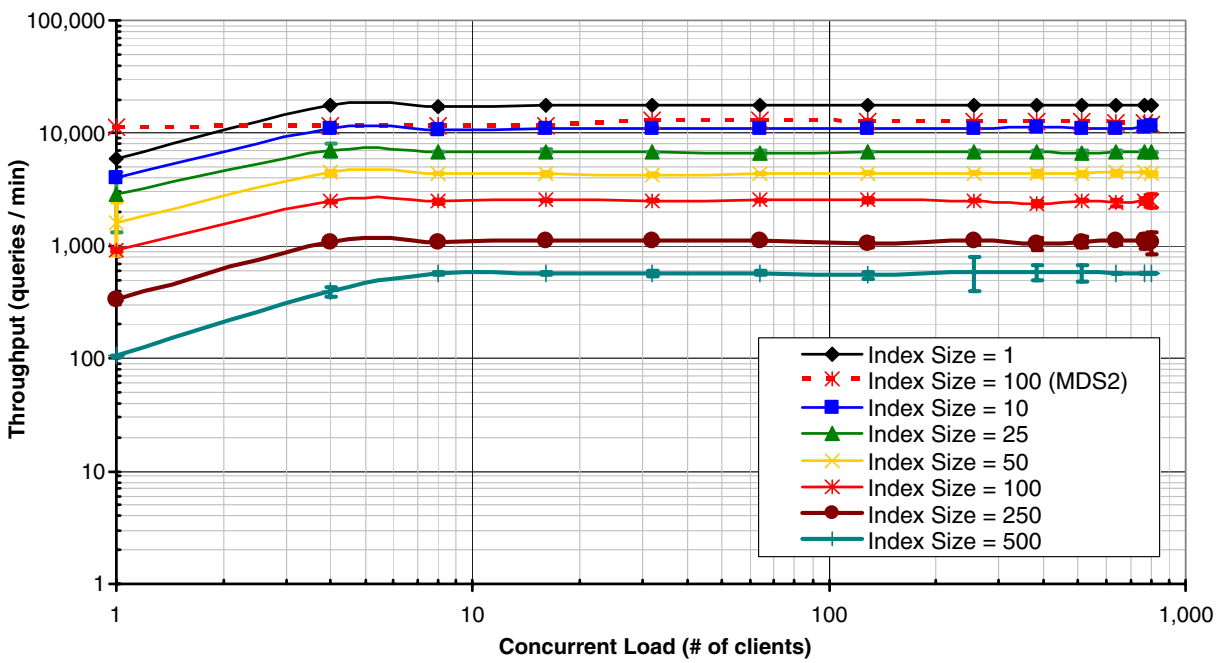

Figure 15 MDS4 Index service throughput performance. Larger values are better. Note the log scales on both axes.

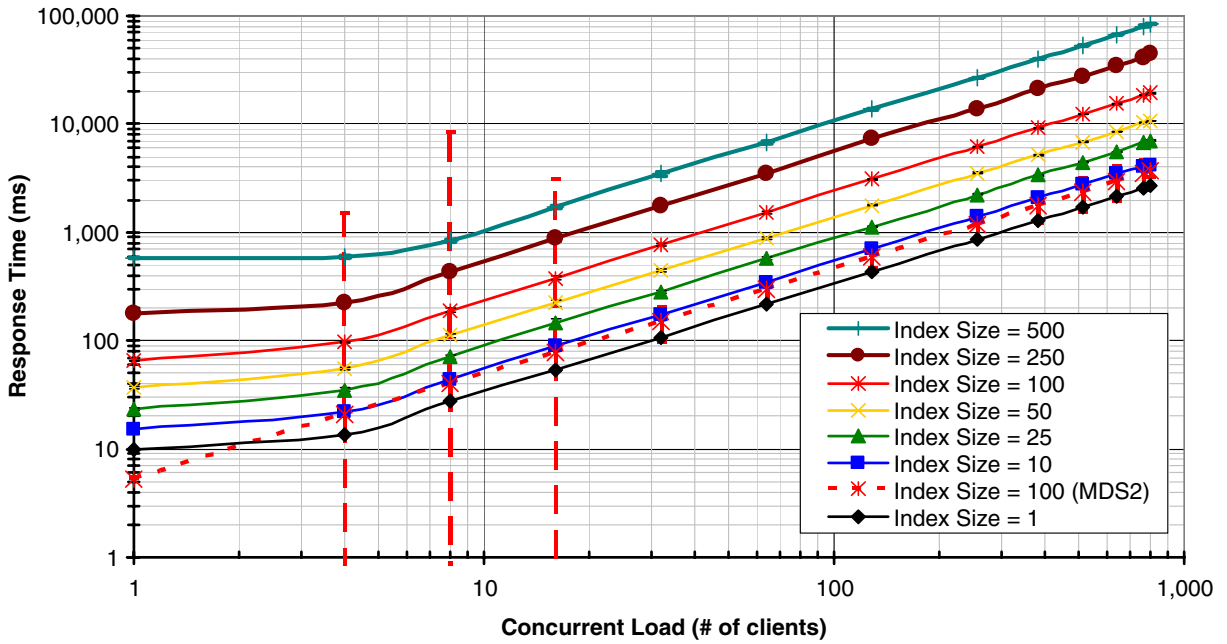

Figure 16 MDS4 Index service response time performance. Smaller values are better. Note the log scales on both axes.

links throttled at maximum 10 MBps and have network latencies between 20 ms and 200 ms (average of 60 ms) to the node hosting the service. On the other hand, the three machines clients in the same LAN are connected via 1 GBps links with 0.1 ms latency.

For the peak portion of the experiment when all 100 clients were running in parallel, the LAN clients (3% of all clients) obtain about 5% of the service throughput. This service rate is significantly more balanced that that observed in a previous performance study [6] of an earlier MDS implementation where local clients were almost able to starve remote clients.

5.4.2. *WS-MDS Performance in a LAN Environment*

Additionally, we have performed experiments in a more controlled environment, the TeraGrid, where nodes linked by a 1 GBps LAN. The WS-MSD server (from Globus Toolkit v4.0.1) and all clients ran on a dual Intel P4 Xeon node with 1 GB of memory. Unlike in all previous experiments where we ramped up load incrementally in this experiment we ran the concurrent load sustained at several discrete points. Results in Figures 15 and 16 are presented in the different manner than other results presented in this paper: Here the *x*-axis presents the offered load (i.e., the number of concurrent clients) while the *y*-axis presents measured performance (either throughput or response time).

Figure 15 presents WS-MDS v4.0.1 throughput for various index sizes (1, 10, 25, 50, 100, 250, and 500) and the performance of an older MDS version shipped with Globus Toolkit v2 (labeled the MDS2 in the figure) and configured with 100 entries for comparison purpose. WS-MDS performance is consistent and robust, as seen by the flat throughput achieved once the service is saturated, regardless of the number of concurrent clients. Figure 16 presents response times, which, for WS-MDS v4.0.1, are also stable and consistent once the service is saturated.

6. Summary and Future Work

Multiple challenges make distributed performance measurements difficult: a) *accuracy* – which implies synchronizing a distributed measurement

platform that might have large communication latencies, b) *flexibility* – to deal with heterogeneity often found in large, geographically distributed environments, c) *scalability* – the to coordinate a large measurement platform, and d) *performance* – to process the high rate of performance measurement transactions generated. In attempting to address these issues, and to simplify and automate service performance evaluations, we have developed DiPerF, a DIstributed PERformance testing Framework.

DiPerF coordinates a pool of machines that test a networked target service (e.g., a Grid service), collects and aggregates performance metrics (e.g., throughput or service response time) from the client point of view, and generates new performance statistics (e.g., on service 'fairness' when serving multiple clients concurrently or on service saturation point). Using DiPerF, we have analyzed the performance characteristics of several components of the Globus Toolkit in wide-area as well as in local networks aiming to better understand their performance in realistic deployment environments. Additionally, using the data collected using DiPerF, it is possible to build predictive models that accurately estimate service performance as a function of service load.

The main contribution of our work is the DiPerF framework itself. DiPerF has been automated so that once deployed it will: a) check what machines nodes are available for testing; b) deploy the client code on the available machines; c) perform time synchronization; d) run the client code in a controlled fashion and collect performance metrics; e) stop and clean up the client code at the end of the experiment; f) aggregate the performance metrics and summarize the results; and g) generate graphs to present the performance characteristics of the tested service.

Additionally, we have also contributed towards a better understanding of various Globus Toolkit components such as GRAM, MDS, and GridFTP services. Using DiPerF in the past 18 months, Globus developers have been able to quantify the performance tradeoffs of various implementation choices and to evaluate the performance of Globus Toolkit components when accessed from a large, geographically distributed client base, a task that would have been tedious and time consuming without DiPerF. These stress tests uncovered obscure bugs that would not have surfaced without the ability to perform controlled large scale testing.

Grids will continue their significant growth and will realize their potential as large-scale infrastructures to support shared resource usage only if efficient usage can be obtained at future large scale. Efficient resource usage, however, is a non-trivial problem in large, dynamic, distributed environments. Today, most Grid services and software are designed and characterized largely based on the designer's intuition and on *ad hoc* experimentation. We believe that methodologies and tools to automatically characterize Grid service deployments' performance, accurate performance models, and algorithms and negotiation protocols that make use of this information to make better informed resource management decisions are key to future Grids success.

We have demonstrated that DiPerF can be used to extract and model service performance characteristics in a client/server scenario. We believe this work can be extended to automate performance characteristics extraction for more complex distributed applications and services. DiPerF can be used to build dynamic performance models to automatically map raw hardware resources to the performance of a particular distributed application and its representative workload. In essence, these dynamic performance models can be thought of as job profiles used for informed resource management. The capability to automatically map complex, multi-dimensional performance requirements and service characteristics among resource providers and consumers is a necessary step to ensure consistent high resource utilization. Automatic matching between the software characterization and a set of raw or logical resources is a much needed functionality that is currently lacking in today's Grid resource management infrastructure.

Acknowledgments Part of this work was carried out while Catalin Dumitrescu and Matei Ripeanu were graduate students with the Distributed Systems Laboratory at The University of Chicago. We also thank the reviewers for the excellent feedback witch allowed us to significantly improve the quality of this paper.

References

1. Peterson, L., Anderson, T., Culler, D., Roscoe, T.: A blueprint for introducing disruptive technology into the internet. In: Proceedings of the First ACM Workshop on Hot Topics in Networking (HotNets), October 2002
2. Foster, I., et al.: The Grid2003 production Grid: Principles and practice. In: 13th IEEE Intl. Symposium on High Performance Distributed Computing, 2004
3. Foster, I., Kesselman, C.: The Grid 2: Blueprint for a New Computing Infrastructure, Chapter 1: Perspectives. Elsevier (2003)
4. Dumitrescu, C., Foster, I.: GRUBER: A Grid Resource SLA-based Broker. EuroPar (2005)
5. Dumitrescu, C., Raicu, I., Foster, I.: DI-GRUBER: A Distributed Approach for Grid Resource Brokering. SC (2005)
6. Raicu, I.: A performance study of the Globus Toolkit® and Grid Services via DiPerF, an automated DIstributed PERformance testing framework. University of Chicago, Computer Science Department, MS thesis, May 2005
7. Raicu, I.: 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
8. Dumitrescu, C., Raicu, I., Foster, I.: Experiences in running workloads over Grid3. The 4th International Conference on Grid and Cooperative Computing (GCC'05), Beijing China, December, 2005
9. Allcock, W., Bresnahan, J., Kettimuthu, R., Link, M., Dumitrescu, C., Raicu, I., Foster, I.; "The Globus Striped GridFTP Framework and Server," sc, p. 54, ACM/IEEE SC 2005 Conference (SC'05), 2005
10. Dumitrescu, C., Raicu, I., Ripeanu, M., Foster, I.: DiPerF: An automated DIstributed PERformance testing Framework. 5th International IEEE/ACM Workshop in Grid Computing, Pittsburg, PA, 2004
11. Internet Protocol Performance Metrics. http://www.advanced.org/ippm/index.html, 1998
12. Paxson, V., Almes, G., Mahdavi, J., Mathis, M.: Framework for IP performance metrics. IETF RFC 2330, 1998
13. The Surveyor Project. http://www.advanced.org/csgippm/, 1998
14. Keynote Systems Inc. http://www.keynote.com, 1998
15. Steigner, Ch., Wilke, J.: Isolating performance bottlenecks in network applications. In: Proceedings of the International IPSI-2003 Conference, Sveti Stefan, Montenegro, October 4–11, 2003
16. Lowekamp, B.B., Miller, N., Karrer, R., Gross, T., Steenkiste, P.: Design, implementation, and evaluation of the Remos Network Monitoring System. Journal of Grid Computing **1**(1), 75–93 (2003)
17. Barford, P., Crovella, M.E.: Measuring Web performance in the wide area. Performance Evaluation Review, Special Issue on Network Traffic Measurement and Workload Characterization, August 1999
18. Banga, G., Druschel, P.: 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
19. Gunter, D., Tierney, B., Tull, C.E., Virmani, V.: On-Demand Grid Application Tuning and Debugging with the NetLogger Activation Service, 4th International Workshop on Grid Computing, Grid2003, Phoenix, Arizona, November 17th, 2003
20. Tsouloupas, G., Dikaiakos, M.: GridBench: A tool for benchmarking Grids. 4th International Workshop on Grid Computing, Grid2003, Phoenix, Arizona, November 17th, 2003
21. The Globus Alliance. Globus Toolkit 3.0 Test Results Page. http://www-unix.globus.org/ogsa/tests/gt3_tests_result.html
22. 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
23. Lee, C., Wolski, R., Foster, I., Kesselman, C., Stepanek, J.: A Network Performance Tool for Grid Environments, SC '99
24. Wolski, R., Spring, N., Hayes, J.: The network weather service: A distributed resource performance forecasting service for metacomputing. Future Generation Computing Systems, 1999
25. select(2) – Linux man page. http://www.die.net/doc/linux/man/man2/select.2.html
26. Chandra, A., Mosberger, D.: Scalability of Linux event-dispatch mechanisms. In: Proceedings of the USENIX Annual Technical Conference (USENIX 2001), Boston, Massachusetts, June 2001
27. Minar, N.: A survey of the NTP protocol. MIT Media Lab, 1999, http://xenia.media.mit.edu/~nelson/research/ntp-survey99
28. Williams, T., Kelley, C.: gnuplot, an interactive plotting program. http://www.gnuplot.info/docs/gnuplot.pdf
29. Massie, M.L., Chun, B.N., Culler, D.E.: The ganglia distributed monitoring system: Design, implementation, and experience. Parallel Comput. *30*(7), (July 2004)
30. Grid3. http://www.ivdgl.org/grid3/
31. Chun, B., Culler, D., Roscoe, T., Bavier, A., Peterson, L., Wawrzoniak, M., Bowman, M.: PlanetLab: An overlay testbed for broad-coverage services. ACM Computer Communications Review, vol. 33, no. 3, July 2003
32. Czajkowski, K., Fitzgerald, S., Foster, I., Kesselman, C.: Grid information services for distributed resource sharing. In: Proceedings of the Tenth IEEE International Symposium on High-Performance Distributed Computing (HPDC-10). IEEE (August 2001)
33. TeraGrid, http://www.teragrid.org/