

Performance Analysis of the Globus Toolkit Monitoring and Discovery Service, MDS2

Xuehai Zhang
Department of Computer Science
University of Chicago
hai@cs.uchicago.edu

Jennifer M. Schopf
Mathematics and Computer Science Division
Argonne National Laboratory
jms@mcs.anl.gov

Abstract

Monitoring and information services form a key component of a distributed system, or Grid. A quantitative study of such services can aid in understanding the performance limitations, advise in the deployment of the monitoring system, and help evaluate future development work. To this end, we examined the performance of the Globus Toolkit® Monitoring and Discovery Service (MDS2) by instrumenting its main services using NetLogger. Our study shows a strong advantage to caching or prefetching the data, as well as the need to have primary components at well-connected sites.

Keywords

Globus Toolkit Monitoring and Discovery Service, Grid Information Services, Performance Analysis

1. Introduction

Grid platforms [FK03] depend on monitoring and information services to support the discovery and monitoring of the distributed resources for various tasks. In-depth studies are needed to understand any performance limitations in common settings.

In our previous work [ZFS03], we investigated the behaviors of the Globus Toolkit Monitoring and Discovery Service (MDS2) [CFF+01,MDS], the most common monitoring system currently used for production Grids, with the focus on analyzing the end-to-end performance of a user request at a very coarse grain. To better understand the unexplained behaviors we saw in that study, in this work we examine MDS behavior at a finer granularity, both by using NetLogger [TG98,TJC+03] technologies to instrument MDS2 server and client codes and by running experiments to evaluate the effect of a large number of concurrent users accessing the different services.

2. MDS2

The Monitoring and Discovery Service [CFF+01,MDS] is built on top of the Lightweight Directory Access Protocol (LDAP) (v3) [Ope]. It is used in the Globus Toolkit [Glo] primarily to address the resource selection problem, namely, how a user identifies the host or set of hosts on which to run an application. MDS2 provides a uniform, flexible interface to data collected by lower-level information providers. It has a decentralized structure that allows it to scale, and it can handle static or dynamic data.

MDS2 has a hierarchical structure that consists of three main components. A Grid Index Information Service (GIIS) provides an aggregate directory of lower-level data. A Grid Resource Information Service (GRIS) runs on a resource and acts as a modular content gateway for a resource. Information providers (IPs) interface from any data collection service and then talk to a GRIS. Each service registers with higher-level services using a soft-state protocol that allows dynamic cleaning of dead resources. Each level also has caching to minimize the transfer of unstale data and lessen network overhead.

We used NetLogger to instrument both the MDS2 server and client codes. NetLogger [TG98,TJC+03] is a toolkit developed by Lawrence Berkeley National Laboratory to monitor, under actual operating conditions, the behavior of elements of a complex distributed system in order to determine exactly where time is spent within such a system and identify the performance bottlenecks. With NetLogger, the components of a distributed system can be modified to produce time-stamped logs of interesting events at all the critical points of the system, which are then correlated to allow detailed characterization of the performance of all aspects of the system. To instrument an application to produce event logs, the application developer inserts calls to the NetLogger API at all the critical points in the code, then

Table 1: The seven phases of an MDS2 query

Phase Name	Phase Definition	Instrumentation Location
<i>Client-Connect</i>	Stage for MDS client program to open a connection to MDS server	Client side
<i>Client-Bind</i>	Stage for MDS client to authenticate to MDS server	Client side
<i>Server-InitSearch</i>	Stage for MDS server performs search initialization	Server side
<i>Server-SearchIndex</i>	Stage for MDS server search indexes for entries	Server side
<i>Server-Invoking</i>	Stage for MDS server to invoke reported information providers or GRIS to generate fresh data	Server side
<i>Server-GenResult</i>	Stage for MDS server to build the results	Server side
<i>Client-EndConnect</i>	Stage for MDS client to receive results and disconnect	Client side

links the application with the NetLogger library. NetLogger is a lightweight tool and adds little overhead to an existing program when used appropriately [TG98].

By adding NetLogger calls we divided the end-to-end path of a MDS2 request into seven phases: (1) *Client-Connect*, (2) *Client-Bind*, (3) *Server-InitSearch*, (4) *Server-SearchIndex*, (5) *Server-Invoking*, (6) *Server-GenResult*, and (7) *Client-EndConnect*, as shown in Table 1. Phases 1, 2, and 7 constitute the MDS2 client side components, and phases 3–6 constitute the server-side components. A NetLogger view of the behaviors of v2.2 and v2.4 MDS2 GRISes accessed by 10 concurrent users is given in Figure 1.

3. MDS2 Performance Results

In this section, we discuss the experiments conducted to test MDS2. First we briefly talk about experimental setup, and then we describe the metrics we used in the experiments. Finally we analyze the performance results.

3.1. Experimental Setup

We ran our experiments between two sites: the Lucky testbed at Argonne National Laboratory (ANL), which provided the MDS2 server-side services, and a testbed at the University of Chicago (UC), which provided the client-side services.

The Lucky testbed we used comprised seven Linux machines with hostnames *lucky{0,1,3,...,7}.mcs.anl* (*lucky2* was unavailable during the experiments) and a shared file system on a 100 Mbps LAN. Each machine was equipped with two 1133 MHz Intel PIII CPUs (with a 512 KB cache per CPU) and 512 MB RAM. *Lucky0* and *lucky6* ran Linux kernel 2.4.10 and the rest ran kernel 2.4.19.

The UC client-side hosts comprised a cluster of 20 Linux machines with a shared file system on a 100 Mbps LAN. Fifteen of them were equipped with a 1208 MHz CPU and 256 MB RAM, while the rest had a slightly slower CPU (but at least 756 MHz), also with 256 MB

RAM. Each machine ran Linux kernel 2.4.17 or a higher version.

The bandwidth between ANL and UC was around 55 Mbits per sec on average (as measured by Iperf [Ipe]), and the latency (Round-Trip Time) was approximately 2.3 msec on average.

We deployed MDS 2.2 and 2.4 on both sites and used NetLogger v2.0.13 to instrument the server and client codes of both versions. To synchronize the clock, we ran NTP 4.1.2 at both the Lucky testbed and UC client hosts.

In our experiments, we simulated up to 600 users querying the MDS2 services simultaneously for 10 minutes, with a waiting period of one second between receiving a request response and issuing the next response, by running individual user processes (scripts) on client machines. We evenly distributed the simulated users to all twenty machines to balance the load.

We used Ganglia [Gan], a cluster monitoring system developed at UC Berkeley, to collect the performance data at five-second intervals. The values reported in each experiment are the average over all the values recorded during a 10-minute time span. We performed all the experiments in a LAN setting to ensure that the performance of the service was affected primarily by the service components and not by other external factors.

3.2. Performance Metrics

We used five performance metrics: throughput, observed response time (*ORT*), request processing time (*RPT*), load1, and CPU-load.

Throughput is defined as the average number of requests (or queries) processed by an MDS2 service component per second.

ORT, equivalent to the metric response time used in our previous work [ZFS03], denotes the average amount of time (in seconds) from the point a user sends out a request till the user gets the response back; it is calculated at the client side. *RPT* is defined as the average time spent at the server side for a MDS2 service to handle a user

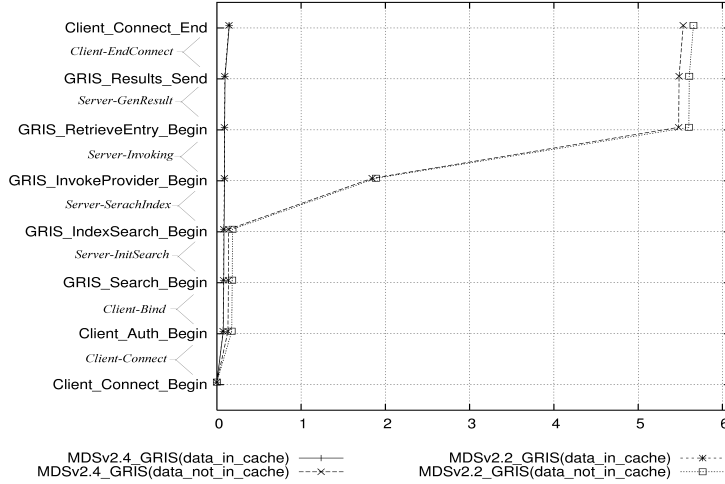


Figure 1: The Seven Phases of an MDS2 Query, Shown Using NetLogger Instrumentation

request. ORT is always greater than RPT , and their relationship can be represented by

$$ORT = T_{Client-Connect} + T_{Client-Bind} + RPT + T_{Client-EndConnect} \quad (1)$$

where $T_{Client-Connect}$, $T_{Client-Bind}$ and $T_{Client-EndConnect}$ denote the time spent on the *Client-Connect* phase, the *Client-Bind* phase, and the *Client-EndConnect* phase respectively. As shown in Table 1, the server side consists of four phases that timewise result in RPT . Therefore, Equation 1 can be expanded to

$$ORT = T_{Client-Connect} + T_{Client-Bind} + RPT + T_{Client-EndConnect}$$

$$RPT = T_{Server-InitSearch} + T_{Server-SearchIndex} + T_{Server-Invoking} + T_{Server-GenResult} \quad (2)$$

We also used two load metrics for the experiments, a one-minute load average (load1) and CPU-load. Load1 is the average number of processes in the ready queue waiting to run over the last minute measured by the Ganglia metric “load_one.” Usually a system is considered overloaded if the load1 value is greater than 3. CPU-load indicates the percentage of the CPU cycles spent in user mode and system mode, which we measured by averaging the sum of `cpu_user` and `cpu_system` recorded by Ganglia. CPU-load may be high while load1 is low if a machine is running a small number of compute-intensive applications. CPU-load may be low while load1 is high if the same machine is trying to run a large number of applications that are blocking on I/O.

3.3. MDS2 Information Server Scalability

As the information server of MDS2, the GRIS can be heavily queried by users. Therefore, in our first set of experiments we evaluated its performance when it was accessed by a large number of users concurrently.

For each MDS2 version, we ran a GRIS on *lucky7*, which had ten information providers reporting to it. We

examined two different scenarios: the GRIS always caching the data from the information providers and the GRIS never caching the data. Our intention was to understand the GRIS performances under two extreme conditions, in order to help us estimate the performance of the average case, which is somewhere between these two. Each query requested all the data elements in the GRIS directory, and this data was generated by all the reported information providers. The average size of requested data was less than 10 KB.

Figures 1–5 show the performance results of v2.2 and v2.4 MDS2 GRISes, for two scenarios described above. The end-to-end performance results are presented in Figures 2–5; Figure 1 shows the NetLogger instrumentation results when each version of the GRIS is accessed by 10 concurrent users.

The results show that a GRIS configured with data caching can achieve a much higher scalability and end-to-end performance (throughput performance in Figure 2 and ORT performance in Figure 3, respectively) than a GRIS without data caching, as seen in our previous work [ZFS03]. Since this work examined the MDS2 behavior in more detail, we were able to determine the performance results of the individual phases for each scenario (shown in Figure 1). We found that the RPT occupies more than 90% of the ORT when a GRIS doesn’t cache data. The much longer delay in the *Server-Invoking* phase is the source of the degraded performance. Since *Server-Invoking* is the stage in which a GRIS invokes the reported information providers to get the data, we believe the delay is caused by the fact that the cost to execute information providers can be high. To make the delay even worse, concurrent queries asking information from the same information provider must compete with each other, since a GRIS can serve them only serially.

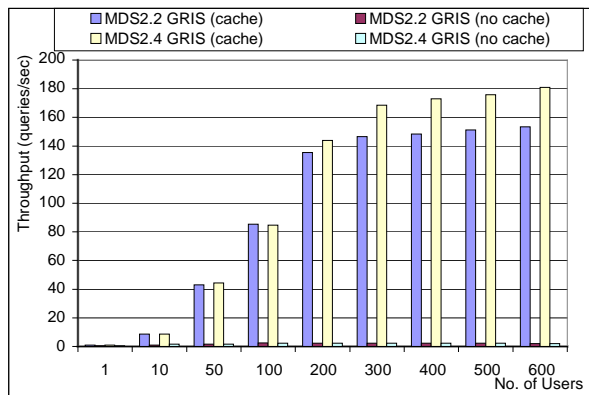


Figure 2: GRIS Throughput vs. Number of Concurrent Users

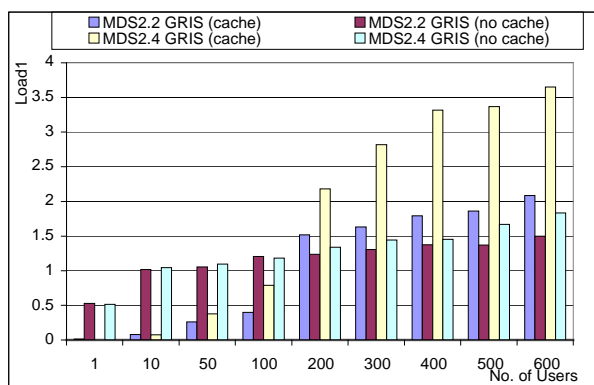
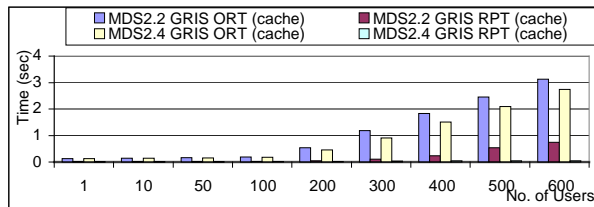


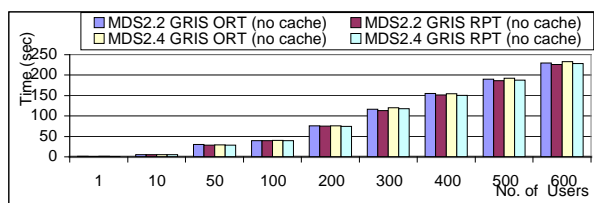
Figure 4: GRIS Host Load1 vs. Number of Concurrent Users

For the GRIS with data caching, the *ORT* did not exceed 3 seconds, compared with a maximum *ORT* of 190 seconds for a GRIS without data caching. A GRIS can serve the concurrent queries with data in its cache rather than invoking low-level information providers; moreover, all cached data can reside in memory to further improve the efficiency. Figure 1 confirms the *Server-Invoking* phase, and the *RPT* is no longer the source of performance bottlenecks.

We observe, however, that the throughput did not follow a constantly increasing rate after the point of 200 concurrent users for both versions of GRIS (Figure 2). This effect is the result of the longer delay of the *Client-Connect* time. Unlike the GRIS without data caching, the performance depends on the *Client-Connect* time: the sum of *Client-Connect* time and *Client-EndConnect* time is about 95% of *ORT*. Many factors may attribute to the delay of *Client-Connect*, for example, network limitations



(a) MDS2 GRIS (with data caching)



(b) MDS2 GRIS (without data caching)

Figure 3: GRIS *ORT* and *RPT* vs. Number of Concurrent Users

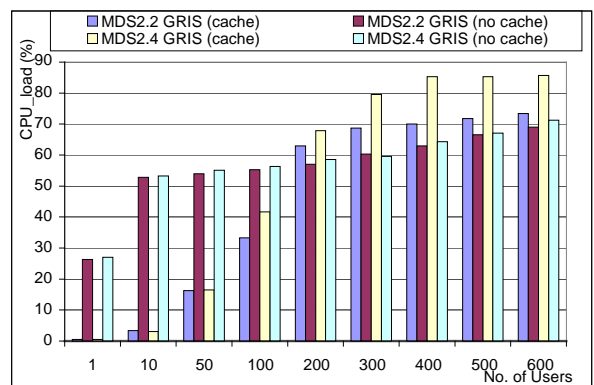


Figure 5: GRIS Host CPU-Load vs. Number of Concurrent Users

at the server side, the operating system's scheduling delay, or the constraint of the LDAP protocol used by MDS2.

Generally, a GRIS experienced a higher load (load1 results in Figure 4 and CPU-load results in Figure 5) with the increasing number of users, whether or not it caches data for each version. This is because more concurrent queries contest for CPU to acquire the service of the GRIS. However, the machine hosting a GRIS without data caching presents a lower load than a machine hosting a GRIS with data caching, indicating that in the former case many of the processes were blocked waiting for resources.

We also observed differences between v2.4 GRIS and v2.2 GRIS. Generally v2.4 GRIS outperforms v2.2 GRIS in the efficiency of the processing requests, due to improved performance in the *Server-SearchIndex* phase, especially

with a large number of users. This is likely due to better memory use in v2.4.

We conclude that the overhead for the MDS2 GRIS can be substantially reduced by data caching because invoking the information providers to serve each query can be expensive. We suggest that, in order to provide

good quality of service, a GRIS should always cache the data that is static or is expensive to calculate or fetch. Moreover, a GRIS should support fewer than 100 users if it has to provide fresh data without data caching for each query.

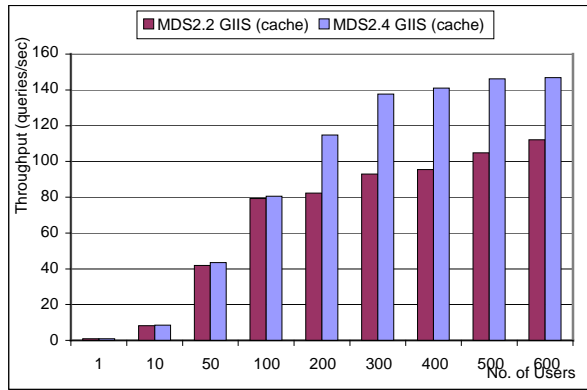


Figure 7: GIIS (with data caching) Throughput vs. Number of Concurrent Users

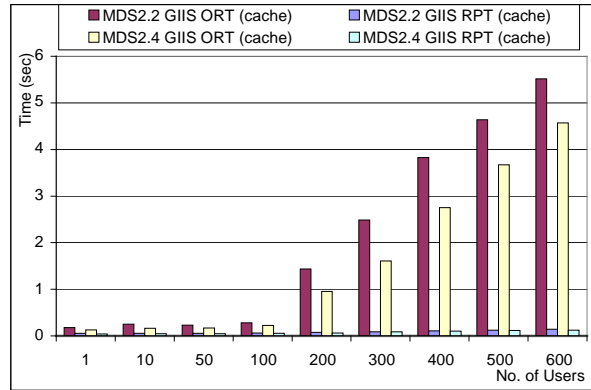


Figure 8: GIIS (with data caching ORT and RPT vs. Number of Concurrent Users

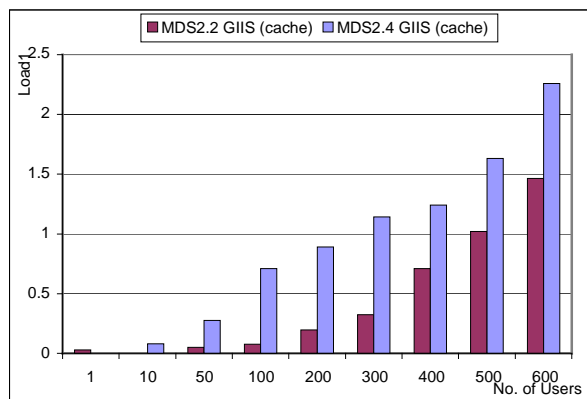


Figure 9: GIIS (with data caching) Host Load1 vs. Number of Concurrent Users

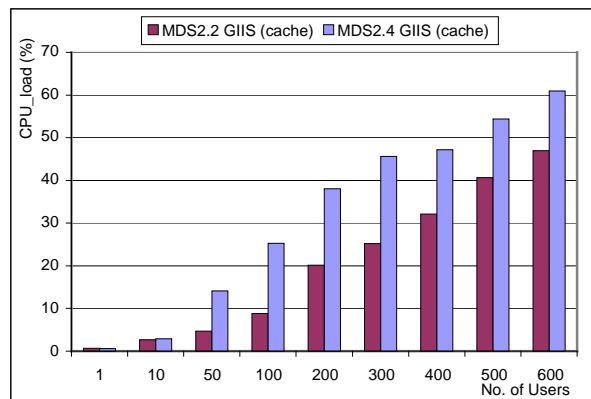


Figure 10: GIIS (with data caching) CPU-load vs. Number of Concurrent Users

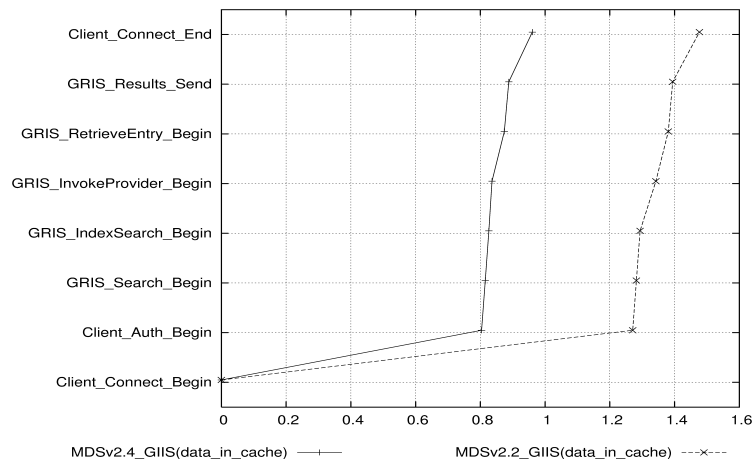


Figure 11: MDS2 GIIS (with data caching) Phases Performance vs. 200 Concurrent Users

3.4. MDS2 Directory Server Scalability

The second functionality of MDS2 we tested was the performance of GIIS as a directory server with the number of concurrent users.

We ran each version of MDS2 GIIS on *lucky1* with a GRIS (containing information from 10 information providers) on each of *lucky3-7* registered to it. To analyze only the directory functionality of the GIIS and not its information serving capacity as an aggregation server, we set the *cachettl* (cache element time to live) parameter to a value larger than 600 seconds to make sure the data was always in the cache during each round of the experiments. Each user queries for all the data elements from the GIIS directory. This means the average data size a query expects is approximately five times bigger than that in the GRIS experiments, about 50 KB.

Figures 7–11 show the performance results of v2.2 and v2.4 MDS2 GIISes with data in their caches. The end-to-end performance results are presented in Figures 7–10; Figure 11 shows the phase performance instrumented by NetLogger when 200 concurrent users access the GIIS.

Similar to our previous work, we found the MDS2 GIIS with data caching scales well and exhibits a high throughput and low *ORT* with respect to the increasing number of users. These results are due to the fact MDS2 GIIS is very efficient in processing the queries at the server side (the *RPT* was always smaller than 0.2 sec) because it does not need to communicate with all the lower-level GRIS to generate the fresh data. We can expect the communication cost to be nontrivial because GIIS and the registered GRIS run on different machines.

The NetLogger instrumentation results shown in Figure 11 also illustrate that the majority of *ORT* is spent on the client side's *Client-Connect* phase for MDS2 GIIS. More concurrent users accessing the same GIIS simply means each user will experience a longer latency in building the connection to the GIIS service on average. Since MDS2 GIIS and GRIS are constructed on nearly the same underlying protocols, we attribute the longer delay of *Client-Connect* time to the same reason we gave to GRIS.

The performance difference of different versions of MDS2 GIIS is also reflected in the results. The v2.4 GIIS shows a higher throughput (Figure 7) and lower *ORT* (Figure 8) than does the v2.2 GIIS when they are accessed by a same number of users. The probable explanation is better use of memory.

Although MDS2 GIIS with data caching can be treated similarly to a GRIS with data caching, their absolute performance is quite different. When accessed by the same number of users, a GRIS is more efficient in serving queries than is the same version of GIIS because

the GIIS has many more entries and the searching takes longer.

From the above experiment we see that using the MDS2 GIIS as a directory server with data caching is a good choice. It can provide good quality of service if serving fewer than 400 users concurrently. With a larger number of users, however, one should duplicate the GIIS in order to keep the quality of service.

4. Conclusions

In this paper, we have investigated the scalability and performance of the Globus Toolkit MDS2 on the fine-grained level. Our present work shows that, when accessed by a large number of concurrent users, both MDS2 GRIS and GIIS present good scalability and performance if they keep data in cache. On the other hand, their performance degrades dramatically without data caching. The NetLogger instrumentation results show that a primary cause of the poor performance is either invoking the reported information provider or consulting the reported GRIS. We also find that the primary components of Grid middleware must be available at well-connected sites, because of the high load seen in the experiments we evaluated.

We plan to do more experiments to address other characteristics of MDS2 GRIS and GIIS with NetLogger instrumentation. For example, we will investigate how the performance of a GRIS scales with the amount of data it contains. We also plan to compare the MDS2 performance with other Grid middleware in the same category, such as R-GMA [CGM+03] and Hawkeye [Haw].

Acknowledgments

We thank both John Mcgee and Ben Clifford at ISI for assistance with the MDS2 and both Brian Tierney and Dan Gunter at LBNL for assistance with NetLogger. We also thank Scott Gose and Charles Bacon for assistance with the testbed at Argonne. This work was supported in part by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under contract W-31-109-Eng-38.

References

[CGM+03] Cooke, A., A. Gray, L. Ma, W. Nutt, J. Magowan, P. Taylor, R. Byrom, L. Field, S. Hicks, and J. Leake, M. Soni, A. Wilson, R. Cordenonsi, L. Cornwall, A. Djaoui, S. Fisher, N. Podhorszki, B. Coghlan, S. Kenny, and D. O'Callaghan, "R-GMA: An Information Integration System for Grid Monitoring." In *Proceedings of the 11th International Conference on Cooperative Information Systems*, 2003.

[CFF+01] Czajkowski, K., S. Fitzgerald, I. Foster, and C. Kesselman, "Grid Information Services for Distributed Resource Sharing." In *Proceedings 10th IEEE International Symposium on High-Performance Distributed Computing (HPDC-10)*, August 2001.

[FK03] Foster, I., and C. Kesselman, eds, *The Grid: Blueprint for a New Computing Infrastructure*, 2nd ed., Morgan Kaufmann, 2003.

[Gan] Ganglia: <http://ganglia.sourceforge.net>.

[Glo] The Globus Alliance: <http://www.globus.org>.

[Haw] Hawkeye: <http://www.cs.wisc.edu/condor/hawkeye>.

[Ipe] Iperf: <http://dast.nlanr.net/Projects/Iperf>.

[MDS] MDS2: <http://www.globus.org/mds/mds2>.

[Ope] OpenLDAP: <http://www.openldap.org/>.

[TG98] Tierney, B., and D. Gunter, "NetLogger Methodology for High Performance Distributed Systems Performance Analysis." In *Proceedings of the 7th IEEE International Symposium on High-Performance Distributed Computing (HPDC-7)*, July 1998.

[TJC+03] Tierney, B., W. Johnston, B. Crowley, G. Hoo, C. Brooks, and D. Gunter, "NetLogger: A Toolkit for Distributed System Performance Tuning and Debugging." In *Proceedings of the 8th IFIP/IEEE International Symposium on Integrated Network Management*, March 2003.

[ZFS03] Zhang, X., J. Freschl, and J. Schopf, "A Performance Study of Monitoring and Information Services for Distributed Systems." In *Proceedings of the 12th IEEE International Symposium on High-Performance Distributed Computing (HPDC-12)*, June 2003.