DGMonitor: a Performance Monitoring Tool for Sandbox-based Desktop Grid Platforms

P. Cicotti[1,2], M. Taufer[1], A. Chien[1]

[1] Department of CSE
University of California, San Diego
taufer, achien@csag.ucsd.edu

[2] Dipartimento di Informatica
Universita di Bologna, Italy
cicotti@cs.unibo.it

Abstract

Accurate, continuous resource monitoring and profiling are critical for enabling performance tuning and scheduling optimization. In desktop grid systems that employ sandboxing, these issues are challenging because (1) subjobs inside sandboxes are executed in a virtual computing environment and (2) the state of the virtual computing environment within the sandboxes is reset to empty after each subjob completes.

DGMonitor is a monitoring tool which builds a global, accurate, and continuous view of real resource utilization for desktop grids with sandboxing. Our monitoring tool measures performance unobtrusively and reliably, uses a simple performance data model, and is easy to use. Our measurements demonstrate that DGMonitor can scale to large desktop grids (up to 12000 workers) with low monitoring overhead in terms of resource consumption (less than 0.1%) on desktop PCs.

Though we developed DGMonitor with the Entropia DCGrid platform, our tool is easily integrated into other desktop grid systems. In all of these systems, DGMonitor data can support existing and novel information services, particularly for performance tuning and scheduling.

Keywords: *Performance monitoring and profiling, sandboxing techniques, distributed computing.*

1 Introduction

Large numbers of high-performance computing applications are being run on desktop grid systems. These systems enable exploitation of unused resources in enterprise intranets and across the Internet, and thus can deliver massive computing power for the study of complex phenomena in a wide variety of scientific fields.

One challenging issue for desktop grid systems is how to achieve good performance. To address efficient usage of networked resources (i.e., computing, storage, and communication resources) and shorten the turn-around time, it is compulsory to know the availability and usage of the real resources in a continuous, global view rather than in an isolated manner. New information services for performance tuning and scheduling optimization of grid systems are under investigation [1, 2]. Ideally, monitoring and profiling tools would provide detailed information with an unobtrusive, continuous, and application independent view for a large number of monitored nodes. In desktop grid environments this is particularly challenging because the desktop PCs are volatile, frequently leaving and joining the desktop grid system, thus making it difficult to locate all the monitored nodes at any time.

Because of the benefits in security and unobtrusiveness, many commercial and open-source desktop grid systems [3, 4, 5] are incorporating sandboxing techniques that isolate grid applications, preventing inappropriate system calls and modifications of the local node environment. The sandboxes make monitoring of desktop grid systems even more challenging because the application computing environment is virtualized and so is the resource availability information. Moreover, sandboxes are typically reset at each subjob termination, preventing continuous monitoring from inside the sandbox [3]. To meet these challenges, we are developing techniques to acquire accurate and continuous monitoring data on desktop grids based on sandboxing techniques.

To address the issues above, we present the architecture of DGMonitor, a monitoring tool to build a global, consistent, and complete view of resource utilization for desktop grids based on sandboxing techniques. Our monitoring tool uses a simple performance data model, provides unobtrusive and reliable performance measures, and is easy to use. To evaluate the quality of DGMonitor, we address and analyze three critical issues for monitoring tools on desktop grids using Entropia DCGrid as a representative model. First, we study the scalability of DGMonitor, exploring the number of desktop PCs that can be monitored accurately, and our results show that DGMonitor can easily monitor grids with 10,000 nodes or more. Second, we quantify the monitoring overhead on each computer. Third, we consider both portability and interoperability of the DGMonitor components and address the possibility of integrating our tool with other

existing information services.
The rest of the paper is organized as follows: In Section 2, we survey existing monitoring systems and motivate why a new monitoring tool like DGMonitor is needed for sandbox-based desktop grids. In Section 3 we present the features and the benefits of sandboxing techniques; we also introduce Entropia DCGrid as a representative of desktop grid systems. In Section 4, we describe the architecture of DGMonitor. Section 5 addresses system metrics, measurement techniques, and communication policies used in DGMonitor for efficient resource monitoring. Section 6 describes how we guarantee unobtrusiveness, Section 7 addresses the system monitoring control, and Section 8 shows how to reconstruct a global performance view from the data collected on local nodes. In Section 9, we measure the scalability and overhead of our monitoring tool. Finally, in Section 10 we address its portability and interoperability.

2 Related Work

To our knowledge, no existing monitoring tools specifically address the challenges of desktop grids based on sandboxes. Well-known monitoring tools for general grid systems, like NWS [6], Ganglia [7] or `dproc` [8], monitor real resource information but at the same time require authenticated access to remote nodes to install monitoring sensors. Reinstalling or restoring such sensors can be a tedious task for highly volatile systems like desktop grids. More generally, such monitoring tools are independent from the grid system itself. This implies the maintenance, control and reliability of two separate infrastructures (one for the monitoring and one for the computing distribution) which need to somehow share common information in real time (e.g., node identifiers, volatility, availability). In contrast, DGMonitor does not require direct authorized access to remote nodes and is easily integrated into the desktop grid infrastructure sharing common information (including authentication) and overcoming the virtualization of resource information from inside a sandboxed computing environment. Finally, DGMonitor operates both on Windows and Linux/Unix systems while the tools listed above only operate on Linux/Unix systems.
More traditional performance monitoring tools like SV-Pablo [9] and Paradyn [10] monitor and visualize the whole distributed system at the application-level isolating sections that are particularly resource demanding. Therefore, these tools normally require some code re-engineering, re-compiling, or re-linking and some embedded instrumentation at the operating system or middleware level. DGMonitor captures both dynamic and static properties with a resource orientation without requiring any intervention at the application level.

3 Desktop Grid Systems

Desktop grids are an emerging class of grid systems which exploit desktop PCs for distributed computing. In general, a central scheduling system accepts subjobs and then distributes them to the desktops PCs that dynamically join and leave the grid, thereby delivering computational power for limited amounts of time. Among the requirements of these systems, security and unobtrusiveness are fundamental issues in guaranteeing desktop grid users that their application data is safe and to the PC owners that their system integrity and data privacy is protected.

3.1 Sandboxing Techniques

Security mechanisms in desktop grids move away from complex authentication mechanisms to self-defense mechanisms based on sandboxing techniques. Sandboxes are virtual, homogeneous computing environments on top of real, heterogeneous distributed environments. A virtual environment acts as an intermediate layer between the subjobs and the operating system increasing the portability of applications, improving security, and providing flexible resource control policies. Running applications inside sandboxes enforces both qualitative and quantitative control of the desktop resources [11], but at the same time it is not possible to directly acquire real system performance data from inside such a virtual computing environment. Sandboxes are based on different techniques which range from interceptions of system calls (e.g., using ptrace on Unix/Linux or dll injection in Windows) to resource control mechanisms at the OS level. The Entropia virtual machine is an example of a user-level sandbox [12]. Sandbox systems like Subterfugue [13] and User Mode Linux [14] are ptrace-based models, whereas Linux Security Module [15] is based on kernel-based models.

3.2 Entropia DCGrid

Entropia DCGrid [3] is a representative of enterprise desktop grid systems based on sandboxing techniques. Entropia DCGrid enables users to run any Win32 application without modifications or special system support. We have chosen Entropia DCGrid as our testbed because of its reliable and robust sandbox implementation. The current version of Entropia DCGrid supports sandboxes on top of Windows 9x/ME/2000/XP.
On the Entropia DCGrid platform, applications are submitted to a job scheduler that distributes subjobs to Entropia clients. Once distributed, the application subjobs run on loosely-coupled workers inside sandboxes. Each Entropia DCGrid client creates one sandbox. The sandbox denies unauthorized access to the underlying systems, ensuring that no system call can inappropriately access the host desktop PC. By acting as a virtual, homogeneous computing environment on top of a real, heterogeneous distributed environment, the sandbox improves the portability of applications and the security for both the host machine and the distributed subjobs. At the end of any application subjob, the content of the sandbox is reset to an initial empty state: the results are sent back to the scheduling master, and the

remaining files are canceled leaving the worker empty and ready to proceed with a new subjob. The Entropia DC-Grid sandbox, under special conditions, allows pinholes to be opened for communication with other sandboxes (e.g., peer-to-peer communication) and for access to the local file system. This aspect does not compromise the security of the whole grid system since the machines running DCGrid clients are located inside the same administrative domain and behind firewalls.

4 Architecture of DGMonitor

Unobtrusiveness and the distributed nature of the grid have led us to design and implement our DGMonitor according to a master-worker scheme. The master acts as a registration authority and controls the overall worker activity of the system. To keep the monitoring of networked resources as unobtrusive as possible, we delegate the collection, the processing, and the storage of performance data to the master. Figure 1 shows the architecture of our lightweight monitoring tool and its processes running on the monitoring master (i.e., `mController, mCollector`) and on the monitored workers (i.e., `mLauncher, mStarter,` and `mDaemon`).

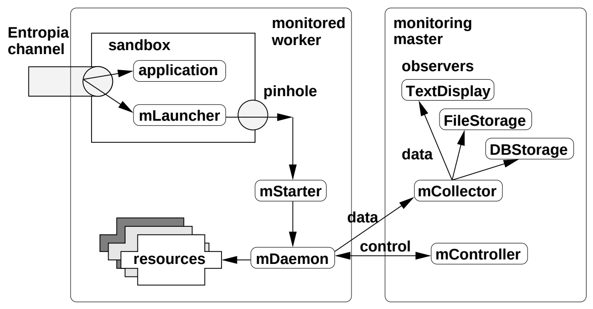

Figure 1. The architecture of the DGMonitor.

We expect the monitored system to change as the desktop grid changes (e.g., new PCs are added or removed). Therefore, it is important that the set of monitored workers is not decided a priory but is continuously updated. To get such a dynamic monitoring approach without requiring any direct authorized access to the worker, we spread the monitoring processes among the PCs of the desktop grid system using the Entropia channel. We do not know a priory which PCs will take part in the distributed computation, but once a desktop PC has received an Entropia client and starts running subjobs, it becomes visible to our DGMonitor. This is achieved by sending tasks to the workers containing both an application subjob and the additional monitoring processes. Since the sandbox is a virtual environment, the monitoring process has to be located outside the sandbox to collect real resource information. It also has to completely break any connection with the sandbox to survive the termination of the application subjob. The sandbox initial empty state is indeed reset once the application subjob has terminated, preventing the monitoring process to continue the monitoring activity if located inside the virtual environment.

On the monitored worker, we use a chain of processes to break all connections with the sandbox and be able to provide a real, continuous stream of performance information to the monitoring master. The `mLauncher` process is started inside the sandbox. It opens a pinhole[1] through the sandbox, creates the `mStarter` process outside the virtual environment, and waits for its termination. The `mStarter` process creates the `mDaemon` process as a stand-alone process and dies immediately. Even if running outside the sandbox, `mStarter` is still intercepted by the virtual environment and if alive at subjob termination, it is detected as an errant child blocking the reset of the computing environment to the initial empty state. The `mDaemon` process is the real monitoring process and samples information about the system resource usage. Before starting its monitoring activity, every `mDaemon` checks the possible existence of a previous instance to avoid duplex `mDaemon` processes on a worker. When the `mDaemon` process is started, any stdout and stderr pipes are closed so that any connection with the sandbox is broken.

We use disjoint control and data channels between monitored workers and the monitoring master. Control packets are sent to and from `mController` through the control channel and carry information to control the execution state of `mDaemons` once the connection with the sandbox is broken. The data channel is used to send resource samples from `mDaemon` to `mCollector`, as shown in Figure 1. DGMonitor uses a relational approach for the data model representation and can store the performance data into relational databases, files, or print it to standard output. Performance data collected by `mCollector` is forwarded to a set of data observers defined in the configuration file of the process (i.e., `TextDisplay, DBStorage, FileStorage`). The chosen observer design pattern makes our monitoring tool highly flexible: so far, the terminal printer implementation (`TextDisplay`) has been used for debugging and testing purposes, whereas, for extensive data collection, we have developed an implementation for connectivity with an MySQL database (`DBStorage`) and for writing to files (`FileStorage`).

5 Efficient Monitoring

5.1 System Metrics

On desktop grid systems, which are non-dedicated systems, the knowledge of dynamic resource properties is vital for improving application performance. DGMonitor is able to capture both dynamic and static properties of the networked resources (i.e., computing and communication resources).

In our approach, we look at each desktop PC as a set of networked resources. The resources under investigation are: CPU, memory, storage devices, and network. For each net-

[1]Pinholes are set through a task configuration file

worked resource, we have identified a limited set of representative metrics capable of capturing the dynamic resource properties. Table 1 reports on the networked resources and the related set of metrics used by DGMonitor. Each metric corresponds to a performance counter that is added to a collective query: at specified frequencies, the system is queried and the metrics are sampled. The data packets used to transmit this information do not exceed 128 Bytes in size.

Resources	Measured metrics
CPU	idle time, user time, system time number of processes, process queue length
Memory	available, cache, cache faults, page faults, page transferred
Storage devices	transfer time, transfer/sec, transfer queue length, Byte/transfer
Network	nic bandwidth, out queue length, packets transferred, packet dropped

Table 1. Networked resources and related metrics.

5.2 Monitoring and Measuring Sensitivity

Dynamic resource properties change at run-time in varying degrees: depending on the use that an application makes of the resources, some resource properties may be updated more frequently than others. Most applications on desktop grid systems are scientific applications which run during long turn-around times ranging from several minutes to hours or days. Therefore, medium-grain (seconds/minutes) and coarse-grain (tens of minutes) measurements of the networked resources give reasonable estimates of system behavior.
Our medium-grain monitoring policy is based on both time- and value thresholds. These time- (δt and Δt) and value thresholds (a vector of resource thresholds $\delta\vec{r}$, one for each of the n metrics monitored reported in Table 1) are set at the beginning of the monitoring process on each worker and can be changed at run-time. Every δt time, mDaemon samples the values of the n metrics, $\vec{r}(t)$, and sends them to mCollector if the following is true:

$$\exists r_i(t) \in \vec{r}(t): \qquad |r_i(t) - r_i(t-\delta t)| > dr_i.$$

To facilitate the accurate rebuilding of performance traces of the networked resources, mDaemon samples the values of the metrics and performs a forced send of the whole metrics vector $\vec{r}$ every Δt time.

5.3 Data Collection

For accurate real-time performance data of a large desktop grid system, frequent and fast monitoring traffic is required. Such kind of transmission may introduce relevant communication intrusions. To obtain low communication intrusions, we use the UDP/IP protocol for transmission of performance monitoring data from mDaemon to mCollector. UDP/IP is a fast, but unreliable transmission protocol (packets may get lost). No attempt to retransmit lost messages is made during our monitoring, but regression models might be used at the data storage level to rebuild lost information [16]. Because UDP/IP does not need to maintain connection states of the monitoring processes, it supports a larger number of monitored workers.
A data message comprises the worker ID assigned during the registration, the timestamp at which the samples have been taken and the several metrics describing the resource properties. The worker ID allows us to simplify the storage procedure of the data: no sophisticated client recognition mechanisms based on the sender IP are required both at the mCollector and data storage levels.

6 Ensuring Self-Cleaning

To survive the sandbox reset at the end of any subjob and to sample real resource values, we allocate the monitoring process, mDaemon, outside the sandbox, thereby violating the unobtrusiveness principle of the sandbox. To guarantee unobtrusiveness, mDaemon is time-bound: DGMonitor uses the concept of *"lease"* or fixed amount of time in which a given mDaemon will be valid for monitoring the resources. The first time a worker joins the pool of monitored desktop PCs and receives an application subjob, the starting mDaemon sends a registration request to mController. If a confirmation for the initial registration is not received by mDaemon, it terminates. At regular intervals ΔT, mDaemon asks for a renewal of the monitoring lease. In case the renewal is not successful, the lease expires and mDaemon terminates. In case of network failure or a failure of mController, the mDaemon processes do not succeed in renewing the lease and terminate. In case of worker failure, no attempt to resume mDaemon is done by the monitoring master.
The choice of the most appropriate ΔT is a critical task that should be based on three main factors: the system scalability, the local overhead due to the renewal of the lease, and the level of unobtrusiveness desired. A small ΔT (e.g., one second) means frequent renewal by a large set of workers and may imply a high load on mController. It also means more frequent use of resources on the worker and the consequent increase of the local overhead. On the other hand, minimizing the lease duration through a small ΔT implies a more responsive mDaemon termination to any renewal failure. In Section 9, we look at how much a small ΔT affects both the scalability and the local overhead.

7 System Monitoring Control

A separate control channel is used to control the behavior of the mDaemon processes, to update the time- and value thresholds, and to register a new lease and keep the registration updated. The loss of a message on this channel might compromise the monitoring process e.g., unexpected termination of a daemon or excessive client intrusion. To assure reliable control communication between master and workers, DGMonitor uses the TCP/IP protocol. Control messages are sent in both directions through the TCP/IP channel.

Master control messages are sent from `mController` to the `mDaemon` processes to control the monitoring (i.e., to start or resume the monitoring activities), to stop or suspend the monitoring, or to kill a monitoring process on a single worker or a group of workers. An update of the time- and value thresholds as well as of the lease duration can also be sent at run-time. Worker control messages are sent from `mDaemon` to `mController` to register a new worker and, at regular intervals, to obtain a renewal of the lease. The `mDaemon` process sends a control message containing the static properties of the networked resources available. It also sends its local time and a set of proposed time- and value thresholds (δt, Δt, ΔT, Δr) based on the static properties of the networked resources. The `mController` process confirms the registration providing `mDaemon` with a worker ID, its global time, and the assigned time- and value thresholds. The global time provided by the monitoring master to the monitored worker is described in detail in the next section.

Master control messages are not frequent and therefore, the related TCP/IP connections do not affect the global system load. On the contrary, registrations or lease renewals might affect the network load for large amounts of workers or small ΔT and, in the worst case, might be responsible for loss of packets in the data channel.

8 Reconstructing Performance from Collected Data

The monitoring master has to be able to patch together the performance data of the workers in a consistent manner. To address this issue, a consistent notion of global time has to be maintained among the several PCs of the desktop grid system. To overcome this problem, a monitoring tool has to introduce some mechanism of synchronization during the performance data sampling. Mechanisms based on ordered events such as barriers for synchronizing the computation and the communication may change the run-time behavior of the monitored application introducing idle times and therefore, change scheduling and execution of the processes.

To cope with the problem of maintaining a global notion of time and at the same time reducing scheduling and execution intrusions, we adopt a notion of global time using synchronizations based on Christian's algorithm [17]. A precise synchronization between master and workers takes place at the registration and at each lease renewal: during these operations the master provides the workers with its local time to which it has added a rough estimation of the time needed for its communication. Figure 2 shows the re-synchronization done at every lease renewal. The `mDaemon` process encloses its timestamp t_0 into the request for the lease renewal. The request is received by `mController` at the time t_1. The `mController` process sends back a confirmation of the lease renewal at time t_2. Both t_2 and $\overline{t_0} = t_0 + (t_2 - t_1)$ are sent back in the confirmation message. At time t_3, the worker receives the renewal and restarts the performance sampling adopting a new global time, $t = t_2 + \frac{t_3 - \overline{t_0}}{2}$. Timestamps of the sample

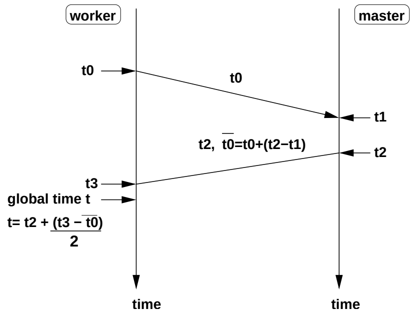
worker
master
t0
t0
t1
t2, $\overline{t0}$=t0+(t2−t1)
t2
t3
global time t
t= t2 + (t3 − $\overline{t0}$) / 2
time
time

Figure 2. Time synchronization between the monitoring master and a monitored worker.

times are delivered in the performance packets containing the sampled performance information.

9 System Evaluation

9.1 Scalability Study

A rapid growth in the number of desktop PCs might imply a significant growth in the number of samples of the resource properties and a consequent increase in terms of overhead at the monitoring master level. To identify when the master becomes a bottleneck to the scalability of the whole system, we measure:

- the maximal number of `mDaemons` collecting data on workers that `mCollector` can serve in real time without losing data packets containing resource properties,
- the maximal number of renewals that `mController` can serve in real time without losing any request from the workers.

Scalability of `mCollector`

To measure the maximal number of `mDaemons` that `mCollector` can serve without losing data packets, we use a multi-threaded packet generator to emulate large numbers of monitored workers sending their metrics samples together. We use real-size data packets of 128 Bytes containing the performance metrics listed in Table 1. We consider two different commodity master configurations: a slow master with 400MHz CPU speed, 256 MByte memory and 8833 IDE disk (slow master) versus a faster master with 2 GHz CPU speed, 512 MByte and ATA-33 IDE disk (fast master). We also consider three different kinds of observers: an observer which directly writes the data in a local file (`FileStorage`), an observer which uses a MySQL database located on the monitoring master (DBStorage_loc) and an observer which uses a MySQL database on a remote machine (DBStorage_rem). We repeat each monitoring test several times, each time lasting more than five minutes. The amount of data packets sent from the workers to the master ranges from 1000 data packets to 15000 data packets per second.

Figure 3 shows the number of received packets (y-axis) versus sent packets (x-axis) for both the slow and the fast master's configurations when the data is locally saved in a file (`FileStorage`). The master with faster and larger system resources (i.e., CPU, memory, HD) is able to serve a larger number of received packets. The maximum number of packets simultaneously served without any loss is 12000 using a fast master and 4000 using a slow master. Figure 4

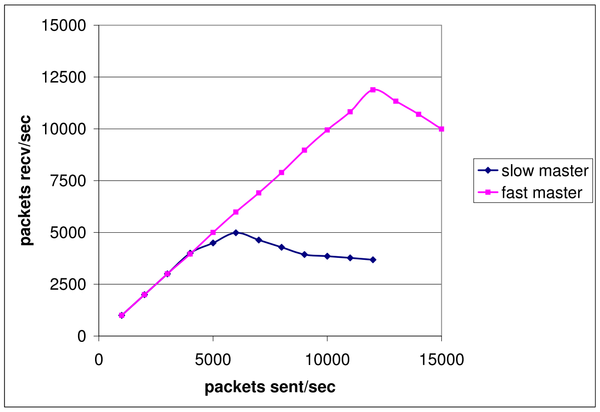

Figure 3. Rate of collected data for different master hardware configurations when the data is directly saved in a local file.

shows the amount of received packets (y-axis) over sent packets (x-axis) using the fast master configuration when the performance data is saved in a local database (DBStorage_loc) and remote database (DBStorage_rem). The remote database is located on a commodity PC with 1.8 GHz CPU speed and 512 MByte RAM. The PC is connected to the monitoring network through a Fast Ethernet connection on a local area network. Saving performance data in a local database gives worse performance in terms of maximum amount of packets stored without any loss than saving this data in a remote database. This is because the local database steals about 30% of the CPU cycles from the master, saturating it in a much faster time. Figure 4 shows that we

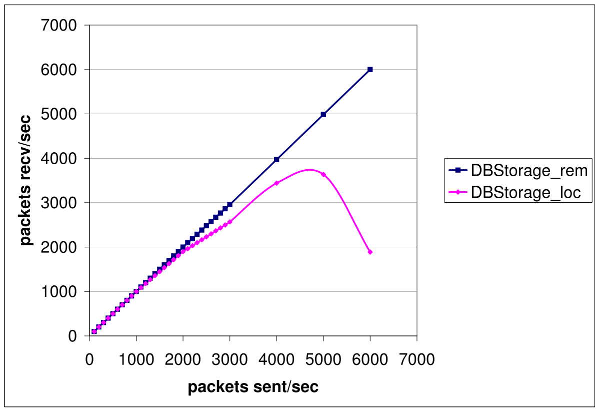

Figure 4. Rate of collected data for different database locations (local and remote) with a fast monitoring master.

receive all the packets up to 1200 packets/sec but we lose more than 14% of the received packets with just 3000 packets/sec. We have found that such a loss increases to 68% when the number of sent packets are 6000 packets/sec.

On the other hand, the storage on the remote database provides better performance in terms of maximum amount of packets stored without any loss. However, for more than 600 packets/sec, such a storage on the remote database is no longer able to take place at run-time because it starts to introduce delays between when the data leaves `mCollector` and when the data is stored into the database. This is mainly due to the congestion control: as a consequence, packets are cached on the master and slowly sent to the database introducing a thrashing phenomena. If the UDP communication between `mDaemon` and `mCollector` allows us to avoid TCP/IP congestion control problems during the collection of the resource metrics, once we want to store this data on a remote database, we have to face such a problem. Moreover, the creation of TCP/IP packets steals CPU cycles additionally affecting the transmission delay. Such a delay increases significantly with the number of packets sent to the database forcing us to reduce the duration of the tests in order to keep the total delay under control and allow the database to store the data in a short time. This is not the case for the local database in which all the data is stored at run-time even for more than 600 packets/sec up to 6000 packets/sec. By running shorter monitoring tests (ranging between 15 and 30 seconds), we observe that we are able to repeat the same performance (in terms of maximum packets received per second without losing any packet) that we have reached when writing the same data on local file.

Scalability of `mController`

To measure the maximum rate of renewals that `mController` can serve, we simulate a large number of workers trying to renew the lease by using a multi-threaded generator. In a first set of tests, the generator is located on one machine and then to increase the load it is replicated on two and three machines. For every renewal, a TCP/IP connection is open and a round trip message is exchanged between the simulated `mDaemon` process and `mController`.

We look at the two different master configurations used in the previous experiment: a monitoring master with 400 MHz CPU speed and 256 MByte memory previously called slow master and a monitoring master with 2 GHz CPU speed and 512 MByte called fast master. Table 2 shows the maximal number of requests served per second for the two configurations. When we try to generate higher

	fast master	slow master
renewal/sec	1440	503

Table 2. Maximal number of renewal per second served by the `mController` process when running on different master configurations.

loads, we observe that `mController` is no longer able to serve more requests and consequently the multi-threaded generator, by adapting its speed, is not able to exceed the limits reported in the table.

Result Analysis and Discussion

Having a TCP/IP connected remote database is a viable solution only if the average number of packets per second expected to be served instantaneously is of the order of a few hundred. A local database can instead serve an expected load of about a thousand packets per second which is in our case the preferable solution. Every simulated load reported in this paper can be considered the result of either a small number of workers (of the order of hundreds) forced to send monitoring packets every second or a larger number (of the order of thousands of workers) running an application which is characterized by slow system changes, so that the performance data transmission happens less frequently (of the order of seconds or even minutes). A lease rate limit arises from the use of the TCP/IP protocol but is not as restrictive as the limits found for `mCollector` since a renewal of the lease is not strictly required every second.

By looking at the results reported above, we can conclude that any technology improvement of the monitoring master (i.e., faster CPU, larger memory, faster disk access) can significantly affect the performance of both the `mCollector` and `mController` processes. Moreover, the monitoring master and the database server perform better if located on different, dedicated machines to assure higher scalability, but still the database host remains the limiting factor. Also architecture improvements, like multiple monitoring masters on which the `mCollector` and `mController` processes are built in hierarchical structures, can significantly increase the number of workers served. These issues are currently under investigation.

9.2 Local Overhead

To study the worker overhead of DGMonitor, we compare the CPU, memory and, network usage rate of `mDaemon` versus the total usage rate for the same resources using the Performance Logs and Alerts. The resource usage sampled every second is piped into a log file. The measurements have shown that the CPU and the memory usage of `mDaemon` remain almost insignificant. The network utilization remains under 0.1% even in a condition of maximal load when the metrics are sampled and sent every second and the lease renewal also takes place every second ($\delta t = \Delta t = 1 \Delta T = 1$).

10 System Integration

10.1 System Portability

For our DGMonitor, multi-threading, synchronization constructs, and socket communication have been designed to be portable: all these parts run not only on Win32 machines and Entropia DCGrid but also on other operating systems based on pthreads like Linux/Unix machines and other desktop grid systems like XtremWeb. On the master side, all the monitoring processes are portable. On the worker side, we are currently developing a `/proc`-based `mDaemon` and we plan to insert our DGMonitor into an XtremWeb platform with workers hosted on both Win32 and Linux machines. Once this extension is finished, we plan to extensively use DGMonitor to monitor large hybrid desktop systems composed of XtremWeb workers and Entropia workers.

10.2 System Interoperability

The data scheme currently adopted in our MySQL database is simple and is used for the collection of all the metrics listed in Table 1. Different implementations and database schemes can easily be implemented to collect sets of metrics, to filter, and to observe specific events. Moreover, an ad-hoc observer implementation can be a direct interface to other specific services. Indeed, our DGMonitor can easily interoperate with existing services, e.g., resource location services [18, 19] or forecasting services [6]. It can also be combined with the Round Robin Database Tool [20, 21] and web-based interfaces to maintain and visualize performance data. To locate networked resources at run-time, DGMonitor with its relational database can be used as a back-end for grid information services like RGIS [1] and R-GMA [2].

11 Conclusion

Accurate, continuous system monitoring and profiling is required to support online performance tuning and scheduling optimization. In this paper, we have described DGMonitor, a tool which enables the collection of this data in sandbox-based desktop grids. We have implemented and evaluated the system on the Entropia DCGrid platform, characterizing its unobtrusiveness. Important design elements include minimal resource usage on the monitored workers and incremental update only when relevant system changes occur. New desktop PCs are added to the monitored system as soon as they receive a subjob and they are removed when their lease cannot be renewed.
Our experiments have shown that DGMonitor can monitor up to 12000 workers in real time with full accuracy, and writing the performance data to a file. When the monitoring master writes the performance data in a local database and the highest accuracy for data collection (every second) is used, DGMonitor can simultaneously collect performance packets from up to 1200 workers without any loss of data. Larger numbers of workers can be monitored in real time under lower precision (time intervals of seconds or even minutes per worker), indicating good scalability of our monitoring tool. The design of DGMonitor is not specific to the Entropia DCGrid system. DGMonitor can easily be integrated into different existing information services to provide performance data and create the base for performance tuning and scheduling optimization of complex desktop grid systems. Finally, we are currently extending DGMonitor to guarantee data privacy by adopting an encryption mechanism (i.e., private and public keys).

Acknowledgments

Supported in part by the National Science Foundation under awards NSF EIA-99-75020 Grads and NSF Cooperative Agreement ANI-0225642 (OptIPuter), NSF CCR-0331645 (VGrADS), NSF ACI-0305390, and NSF Research Infrastructure Grant EIA-0303622. Support from Hewlett-Packard, BigBangwidth, Microsoft, and Intel is also gratefully acknowledged. We wish to thank Alex Olugbile and Andre Kerstens for their helpful comments and suggestions.

References

[1] P. Dinda and D. Lu. Nondeterministic Queries in a Relational Grid Information Service. In *Proc. of Supercomputing 2003 (SC 2003)*, Phoenix, AZ, Nov 2003.

[2] S. Fisher. Relational Model for Information and Monitoring. Technical report, Tech. Rep. Informational Draft GWD-GP-7-1, Grid Forum, 2001.

[3] A. Chien, B. Calder, S. Elbert, and K. Bhatia. Entropia: Architecture and Performance of an Enterprise Desktop Grid System. *Journal of Parallel Distributed Computing*, 2003.

[4] D. Anderson et al. United Devices - Building the worlds largest computer, one computer at a time. http://www.ud.com.

[5] O. Lodygensky, G. Fedak, V. Neri, F. Cappello, Thain D., and M. Livny. XtremWeb and Condor : Sharing Resources between Internet connected Condor Pool. In *GP2PC2003 (Global and Peer-to-Peer Computing on Large Scale Distributed Systems) colocated with IEEE/ACM CCGRID2003*, Tokyo Japan, May 2003.

[6] R. Wolski, N. Spring, and J. Hayes. The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing. *Journal of Future Generation Computing Systems*, 15(5-6):757–768, Oct 1999.

[7] M. Massie, B. Chun, and D. Culler. The Ganglia Distributed Monitoring System: Design, Implementation, and Experience. 2003.

[8] S. Agarwala, C. Poellabauer, J. Kong, K. Schwan, and M. Wolf. Resource-Aware Stream Management with the Customizable dproc Distributed Monitoring Mechanisms. In *Proc. of the 12th IEEE International Symposium on High Performance Distributed Computing (HPDC-12)*, Seattle, WA, Jun 2003.

[9] L. De Rose, Y. Zhang, and D.A. Reed. SvPablo: A Multi-language Performance Analysis System. *Lecture Notes in Computer Science*, 1469:352–??, 1998.

[10] B.P. Miller, M.D. Callaghan, J.M. Cargille, J.K. Hollingsworth, R. B. Irvin, K.L. Karavanic, K. Kunchithapadam, and T. Newhall. The paradyn parallel performance measurement tool. *IEEE Computer*, 28(11):37–46, 1995.

[11] F. Chang, A. Itzkovitz, and V. Karamcheti. User-level Resource-constrained Sandboxing. In *4th USENIX Windows Systems Symposium (WSS 2000)*, Seattle, Aug 2000.

[12] B. Calder, A. Chien, J. Wang, and D. Yang. The Entropia Virtual Machine for Desktop Grids. Technical Report CS2003-0773, University of California, San Diego, CSE Technical Report CS2003-0773, 2003.

[13] M. Callman and M. Pavel. http://subterfugue.org/.

[14] J. Dike. A User Mode Port of the Linux Kernel. In *Proc. of the 4th Annual Linux Showcase*, Atlanta, GA, 2000.

[15] C. Cowan and D. Wagner. http://lsm.immunix.org/.

[16] M. Taufer. *Inverting Middleware: Performance Analysis of Layered Application Codes in High Performance Distributed Computing*. PhD thesis, Laboratory for Computer Systems, Department of Computer Science, Swiss Federal Institute of Technology (ETH) Zurich, Oct 2002.

[17] F. Cristian and C. Fetzer. The Timed Asynchronous Distributed System Model. *IEEE Transactions on Parallel and Distributed Systems*, 10(6):642–??, 1999.

[18] I. Foster and C. Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *Intl J. Supercomputer Applications*, 11(2):115–128, 1997.

[19] R. Raman, M. Livny, and M. Solomon. Matchmaking: Distributed Resource Management for High Throughput Computing. In *Proc. of the Seventh IEEE International Symposium on High Performance Distributed Computing (HPDC-7)*, Chicago, IL, Kul 1998.

[20] D. A. Menasce and V. Almeida. *Capacity Planning for Web Services*. Prentice Hall PTR, 2002.

[21] M. Karaul. *Metacomputing and Resource Allocation on the World Wide Web*. PhD thesis, New York University, May 1998.