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

Abstract

Accurate and continuous monitoring and profiling are important issues of performance tuning and scheduling optimization. In desktop grid systems based on sandboxing techniques these issues are particularly challenging because (1) subjobs inside sandboxes are executed in a virtual environment and (2) sandboxes are usually reset to an initial (empty) state at each subjob termination.

To address this problem, we present in this paper DGMonitor, a monitoring tool to build a global, accurate and continuous view of real resource utilization for desktop grids based on sandboxing techniques. Our monitoring tool provides unobtrusive and reliable performance measures, uses a simple performance data model and is easy to use. Our work demonstrates that DGMonitor can easily take over the monitoring of large desktop grids (up to 12000 workers) maintaining low load in terms of resource consumption due to the monitoring process (less than 0.1%) on desktop PCs.

Although we use DGMonitor with the Entropia DGrid platform, it can be easily integrated in other desktop grids and its data can be used as an information source for existing information services for performance tuning and scheduling optimization.

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

1 Introduction

A wide range of high-performance computing applications are ported to widely distributed compute environments like desktop grid systems. Desktop grid computing, exploiting unused resources in the Enterprise and across the Internet, can deliver massive compute power for the study of complex and compute-demanding problems in a variety of different scientific fields.
System performance on desktop grids is still an open issue. To address efficient usage of networked resources (distributed compute and communication resources) and shorten turn-around times of distributed applications, new information services for performance tuning and scheduling optimization are under investigation [10, 11, 7]. In general, for distributed systems, information services need to know and use the resource properties in a continuous, global view rather than in an isolated way. Moreover, any possible optimization requires knowledge of the real resource availability.
Accurate monitoring and profiling tools should provide information services with an unobtrusive, continuous and application independent overview of the real performance behavior for a large number of monitored nodes. In desktop grid environments this is challenging because the desktop PCs frequently leave and join the desktop grid system making it difficult to locate all the monitored nodes at any time. Monitoring and profiling desktop grid systems based on sandboxing techniques is even more challenging because the compute environment for such systems is virtual (and so is the information about the resource availability). Moreover, sandboxes are usually reset to an initial (empty) state at each subjob termination, making it impossible to do any continuous monitoring from inside the sandbox [4]. The numerous advantages of using desktop grid systems based on sandboxing techniques in terms of security and unobtrusiveness makes these systems more and more attractive. Besides security and unobtrusiveness, the interest for accurate system performance measures is also raising in importance, but still no specific performance tool for sandbox-based desktop grids has been developed yet.
As our contribution to the performance issue raised above, we present in this paper 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 our DGMonitor, we address and analyze three critical issues of monitoring tools on distributed systems using Entropia DCGrid as a representative of desktop grid systems based on sandboxing techniques. First, we study the scalability of DGMonitor in terms of number of monitored desktop grid PCs and show how our tool can easily take over the monitoring of large desktop grids. Second, we quantify the overhead in terms of resource consumption on desktop PCs due to the monitoring process. Third, we look at both the portability and interoperability of the DGMonitor components addressing possible integration of our tool in existing information services.
The rest of the paper is organized as follows: In Section 2 we present the features and the benefits of sandboxing techniques and introduce Entropia DCGrid as a representative of desktop grid systems based on such techniques. In Section 3, we describe the architecture of DGMonitor. Section 4 addresses respectively system metrics, measuring techniques and communication policies used in DGMonitor for efficient resource monitoring. Section 5 describes how we guarantee unobtrusiveness, Section 6 addresses the system monitoring control and Section 7 shows how to reconstruct global performance view from the local, collected data. In Section 8, we measure scalability and overhead of our monitoring tool while in Section 9 we address its portability and interoperability. In the end, in Section 10 we look at some relevant related work and conclude.

2 Desktop Grid Systems

Desktop grids are an emerging category of systems for distributed computing based on desktop PCs. In general, a central scheduling system is used to submit subjobs to the grid. The distributed computation takes place on PCs that dynamically join and leave the grid delivering computational power for limited amounts of time. Among the requirements of these systems, security and unobtrusiveness are fundamental keys in guaranteeing desktop grid users that their processed information is safe and providers of the PCs that no malicious action is being taken against their system integrity and data privacy.

2.1 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 environments on top of real, heterogeneous distributed environments. Such 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 can enforce both qualitative and quantitative control of the desktop resources [3] making not possible for a monitoring tool to measure real system performance from inside such a virtual compute 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. Sandbox systems like Subterfuge [2] and User Model Linux [9] are ptrace-based models, Linux Security Module [5] is based on kernel-based models.

2.2 Entropia DCGrid

Entropia DCGrid [4] 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 because of its reliable and robust sandbox implementation. The current version of Entropia DCGrid supports sandboxes on top of Windows ME/XP/2000/9x.
On the Entropia DCGrid platform, applications are submitted to a job submitter whose subjobs are distributed to Entropia clients by a scheduling master. Once distributed, the application subjobs run on loosely-coupled workers inside sandboxes. The sandbox denies unauthorized access to the underlying systems, ensuring that no system call can inappropriately access the host desktop PC. It acts as a virtual, homogeneous compute environment on top of a real, heterogeneous distributed environment helping the portability of applications and improving 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 a new subjob. In Entropia DCGrid, it is allowed to open pinholes for communication with other sandboxes (peer-to-peer communication) and for reaching 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. Each Entropia DCGrid client creates one sandbox. The user can launch two processes inside the same sandbox and can also specify that one process can run without waiting for the other process to terminate.

3 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.

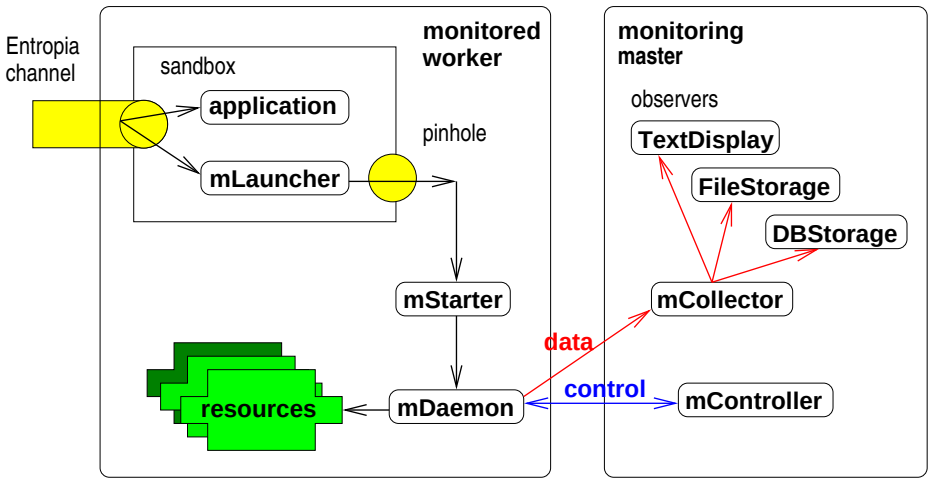

Figure 1: The architecture of the DGMonitor.

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 priori but is continuously updated at run-time. 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 priori which PCs will take part in the distributed computation, but once they have received an Entropia client and start running subjobs, they become 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.
We use a chain of processes (i.e. `mLauncher`, `mStarter` and `mDaemon`) 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, the `mStarter` process is still intercepted by the virtual environment and if alive at subjob termination, it is detected as an errant child blocking the reset to the initial empty state. The `mDaemon` process is the real monitoring

[1]Pinholes are set through a task configuration file

process which 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 workers and the monitoring master. Control packets are sent to and from the `mController` process through the control channel and carry information to control the execution state of the `mDaemon` processes once the connection with the sandbox is broken. The data channel is used to send resource samples from the `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 the `mCollector` process 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 while for extensive data collection we have developed an implementation for connectivity with an MySQL database (DBStorage) and for writing to files (FileStorage).

4 Efficient Monitoring

4.1 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. compute 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 networked 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

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, Bytes/transfer
Network	nic bandwidth, out queue length, packets transferred, packet dropped

Table 1: Networked resources and related metrics.

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.

4.2 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. 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, the `mDaemon` process samples the values of the n metrics, $\vec{r}(t)$, and sends them to the `mCollector` process 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, the `mDaemon` process samples the values of the metrics and performs a forced send of the whole metrics vector $\vec{r}$ every Δt time.

4.3 Data Collection

For accurate real-time performance data of a large desktop grid system, frequent and fast monitoring traffic is required. Such kind of transmissions may introduce relevant communication intrusions. To obtain low communication intrusions, we use the UDP/IP protocol for transmission of performance monitoring data from the `mDaemon` process to the `mCollector` process.
UDP/IP is a fast, but unreliable transmission protocol (packets may get lost). No attempt to retransmit lost messages is made and regression models might be used at the data storage level to rebuild lost information [18]. 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 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.

5 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, the `mDaemon` process is time-bound: DGMonitor uses the concept of a "lease" or 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 the `mController`. If a confirmation for the initial registration is not received by the `mDaemon` process, 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 the mController process, the mDaemon processes do not succeed in renewing the lease and terminate. In case of worker failure, no attempt to resume the 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 the mController process. 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 8 we will look at how much a small ΔT affects both the scalability and the local overhead.

6 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 the 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. Table 2 shows the list of the master control messages and the related action.

Message	Action
STOP	stop an active mDaemon
START	restart a frozen mDaemon
KILL	kill an active mDaemon
UPDATE	update the time and value thresholds (δt, Δt, ΔT, Δr)

Table 2: Master control messages and the related action.

Worker control messages are sent from the mDaemon to the mController to register a new worker and at regular intervals to obtain a renewal of the lease. Figure 2 shows the exchange of control messages during a registration. The mDaemon sends a control message containing

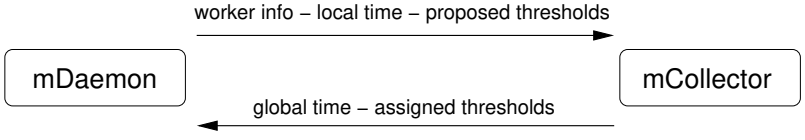

Figure 2: Exchange of control messages between monitoring master and monitored worker during a new registration.

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` confirms the registration providing the `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 and small ΔT and in the worst case might be responsible for loss of packets in the data channel.

7 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 [6]. 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 3 shows the re-synchronization done at every lease renewal. The `mDaemon` encloses into the request for the lease renewal its timestamp t_0. The request is received by the `mController` at the time t_1. The `mController` sends back a confirmation of the lease renewal at time t_2. Both t_2 and $\bar{t}_0 = t_0 + (t_2 - t_1)$ 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, $t = t_2 + \frac{t_3 - \bar{t}_0}{2}$. Timestamps of the sample times are delivered in the performance packets containing the sampled performance information.

8 System Evaluation

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

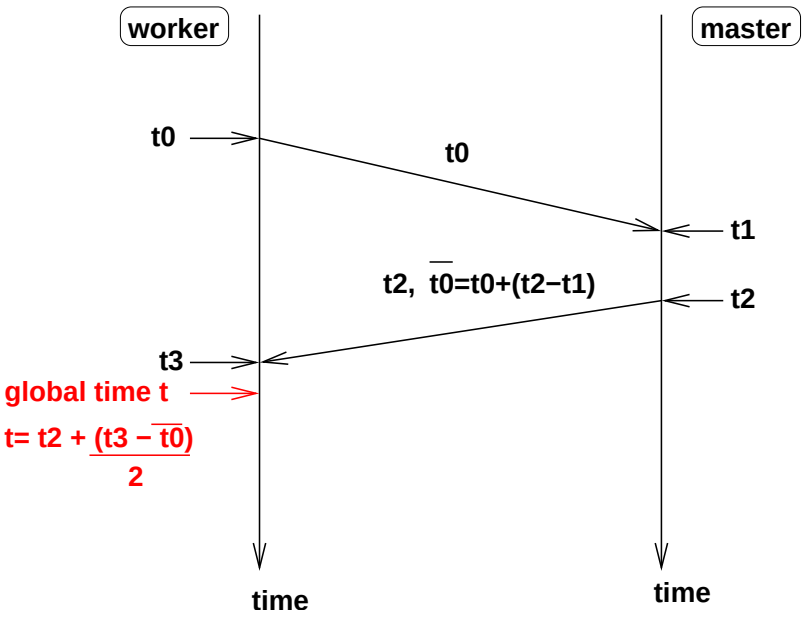

Figure 3: Time synchronization between the monitoring master and a monitored worker.

Scalability of the `mCollector`

To measure the maximal number of `mDaemons` that the `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, 256MBytes memory and 8833 IDE disk (slow_master) versus a faster master with 2GHz CPU speed, 512MB 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 4 shows the number of received packets (y-axis) versus sent packets (x-axis) for both the slow and the fast master's configuration 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 5 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 1.8GHz CPU speed and 512MBytes 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 5 shows that we receive all the packets up to 1200/sec but we lose more than 14% of

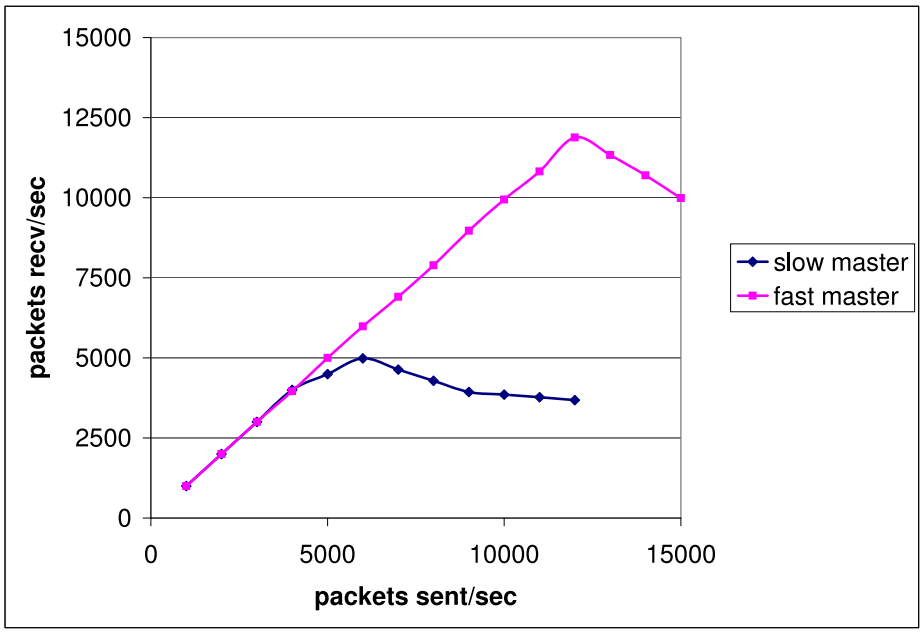

Figure 4: Rate of collected data for different master hardware configurations when the data is directly saved in a local file.

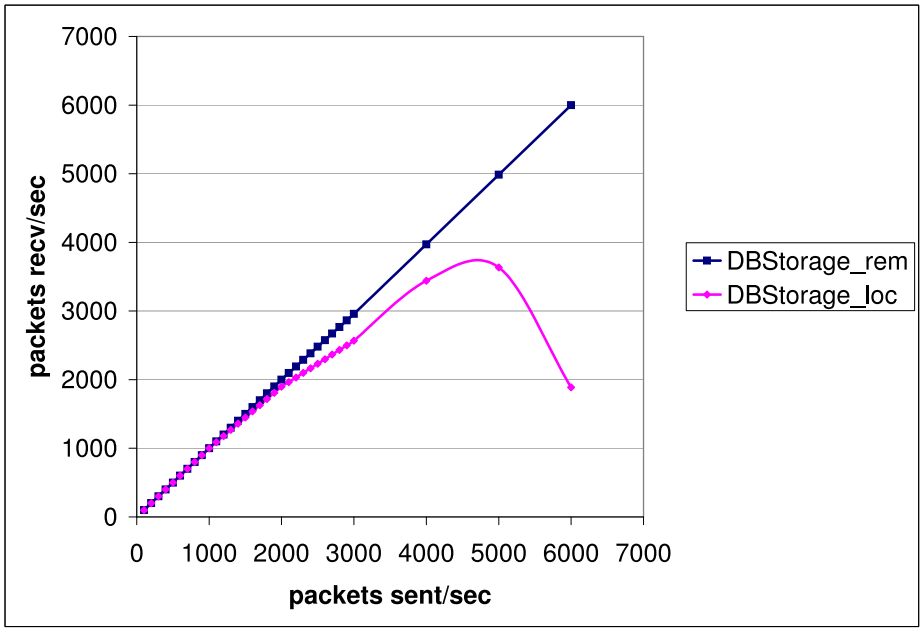

Figure 5: Rate of collected data for different database locations (local and remote) with a fast monitoring master.

the received packets with just 3000 packets/second. We have found that such a loss increases to 68% when the number of sent packets are 6000/second.

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 the `mCollector` and when it 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/second up to 6000 packets/second. 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 the `mController`

To measure the maximum rate of renewals that the `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` and the `mController`.
We look at the two different master configurations used in the previous experiment: a monitoring master with 400MHz CPU speed and 256MB memory previously called slow_master and a monitoring master with 2GHz CPU speed and 512MB called fast_master. Table 3 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 3: Maximal number of renewal per second served by the `mController` process when running on different master configurations.

loads, we observe that the `mController` process 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 in our case is 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 of workers) 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 found arises from the use of the TCP/IP protocol but is not as restrictive as the limits found for the `mCollector` since a renewal of the lease every second is not strictly required.
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 `mCollector` and `mController` processes are built in hierarchical structures can significantly increase the number of workers served. These issues are currently under investigation.

8.2 Local Overhead

To study the worker overhead of DGMonitor, we compare the CPU, memory and network usage rate of the `mDaemon` process 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 ($\delta t = \Delta t = 1 \Delta T = 1$).

9 System Integration

9.1 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.

9.2 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 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 [12, 17] or forecasting services [19]. It can also be combined with the Round Robin Database Tool [15, 13] 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 [10] and R-GMA [11].

10 Related Work

To our knowledge, no existing monitoring tools support desktop grids based on sandboxes as does our DGMonitor. Most of the existing monitoring tools (e.g. SVPablo [8], Paradyn [16]) monitor the whole distributed system at the application level isolating software parts which are particularly resource demanding. Therefore, they normally require some code re-engineering, re-compiling or re-linking and some embedded instrumentation at the operating system or middleware level. Monitoring tools like NWS [19], Ganglia [14] or `dproc` [1] require direct access to Linux/Unix machines. DGMonitor does not require direct authorized access and can run on both Windows and Linux/Unix machines.
Monitoring tools like `dproc` maintain a hierarchical structure of the distributed system and the performance data. To extract performance data from the entire system the user is required to know the hierarchical topology of the monitored system and a hierarchical query language is required. Any change of the hierarchical topology may affect the reconstruction of the global performance view. As for RGIS [10] and R-GMA [11], DGMonitor has an easy data model of the networked resources based on a relational data model rather than hierarchical data model. RGIS monitors mainly static resource properties while R-GMA looks at the dynamic properties. DGMonitor is able to capture both dynamic and static properties of the networked resources.

11 Conclusion

Accurate, continuous system monitoring and profiling allow us to retrieve the data necessary to information services for performance tuning and scheduling optimization. In this paper we have presented DGMonitor, a monitoring tool easily adaptable to different desktop grids and we have proven its applicability on the Entropia DCGrid platform as a representative of sandbox-based desktop grids. The implementation of DGMonitor is unobtrusive thanks to its minimal resource usage on the monitored workers and its efficient monitoring process that keeps the performance information updated periodically and at the occurrence of relevant system changes. New desktop PCs are added to the monitored system as soon as they receive a subjob and they are removed when the lease cannot be renewed.
Our experiments have shown that DGMonitor can monitor up to 12000 workers in real time without losing any performance packets when writing the monitoring data in 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 looser performance collection configurations (of the order of seconds or even minutes per worker), allowing us to claim a good scalability of our monitoring tool. We are currently extending DGMonitor to guarantee data privacy by adopting an encryption mechanism (i.e. private and public keys). Last but not least, DGMonitor can easily be integrated into different existing information services to provide performance data and create the base for performance tuning and scheduling optimization playing a fundamental role in complex desktop grid systems.

References

[1] 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.

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

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

[4] 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.

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

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

[7] H. Dail, F. Berman, and H. Casanova. A Decoupled Scheduling Approach for Grid Application Development Environments. *Journal of Parallel and Distributed Computing*, 63(5):505–524, 2003.

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

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

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

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

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

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

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

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

[16] 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.

[17] 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.

[18] 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.

[19] 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.