J Grid Computing (2006) 4: 225–246
DOI 10.1007/s10723-006-9040-x

Labs of the World, Unite!!!

Walfredo Cirne · Francisco Brasileiro · Nazareno Andrade · Lauro B. Costa · Alisson Andrade · Reynaldo Novaes · Miranda Mowbray

Received: 31 August 2005 / Accepted: 28 February 2006 / Published online: 27 June 2006
© Springer Science+Business Media B.V. 2006

Abstract eScience is rapidly changing the way we do research. As a result, many research labs now need non-trivial computational power. Grid and voluntary computing are well-established solutions for this need. However, not all labs can effectively benefit from these technologies. In particular, small and medium research labs (which are the majority of the labs in the world) have a hard time using these technologies as they demand high visibility projects and/or high-qualified computer personnel. This paper describes OurGrid, a system designed to fill this gap. OurGrid is an open, free-to-join, cooperative Grid in which labs donate their idle computational resources in exchange for accessing other labs' idle resources when needed. It relies on an incentive mechanism that makes it in the best interest of participants to collaborate with the system, employs a novel application scheduling technique that demands very little information, and uses virtual machines to isolate applications and thus provide security. The vision is that OurGrid enables labs to combine their resources in a massive worldwide computing platform. OurGrid is in production since December 2004. Any lab can join it by downloading its software from http://www.ourgrid.org.

Key words free-to-join Grids · Grid computing · Grid scheduling · incentive to collaborate · peer-to-peer Grids · sandboxing

W. Cirne (✉) · F. Brasileiro · N. Andrade ·
L. B. Costa · A. Andrade
Departamento de Sistemas e Computação,
Laboratório de Sistemas Distribuídos,
Universidade Federal de Campina Grande,
Paraiba, Brazil
e-mail: walfredo@dsc.ufcg.edu.br

F. Brasileiro
e-mail: fubica@dsc.ufcg.edu.br

N. Andrade
e-mail: nazareno@dsc.ufcg.edu.br

L. B. Costa
e-mail: lauro@dsc.ufcg.edu.br

A. Andrade
e-mail: aandrade@dsc.ufcg.edu.br

R. Novaes
Hewlett Packard,
Porto Alegre, RS, Brazil
e-mail: reynaldo.novaes@hp.com

M. Mowbray
Hewlett Packard,
Bristol, United Kingdom
e-mail: miranda.mowbray@hp.com

1. Introduction

The recent advances in computing and networking are changing the way we do scientific research, a trend that has been dubbed eScience. Thanks to the power of computer-based communication,

research is now a much more collaborative endeavor. Moreover, computers play an ever-increasing role in the process of scientific discovery. Data analysis without computers sounds antediluvian. Simulation has joined theory and experimentation as the third scientific methodology. As a result, many research labs now demand non-trivial computing capabilities. Buying more computers is a natural answer to this demand. But compute demand seems to be insatiable. No matter how much computing resource is available, it is commonplace to hear “we could do more/better research if we had access to more computing power.”

Computer scientists have long recognized this fact, and began to address it by providing a way to harvest the computing power going idle in one’s lab or university [43]. This represented an important step forward, but has limited scale. Latter, this idea evolved into harvesting the computing power going idle in the Internet [2, 3], in what became known as *voluntary computing*. At about the same time, *Grid computing* [11, 34] appeared with the enticing vision of “plug into the Grid and solve your computational problem.”

Voluntary computing has been able to deliver unprecedented computing power to some applications. For example, at the beginning of March 2005, SETI@home had mustered more than 2.2 million years of CPU time, from over 5.3 million users, spread across 226 countries [57]. However, in order to benefit from voluntary computing, it is necessary to have a high visibility project, set up a large control center to manage the volunteers, and put a lot of effort into ‘publicity’ to convince people to install the worker module. Naturally, being in a prestigious University and having a qualified team for software development and system administration can help a great deal towards this. Alas, these conditions do not hold for most research labs in the world.

On the other hand, Grid computing is turning from promise into reality. There are now a few large Globus-based Grids in production. With dozens of sites and thousands of computers, CERN’s LCG [17] is probably the best current example of what Grids can achieve. However, current Grids are somewhat limited in scale [33, 51], not going beyond dozens of sites. Moreover, installing, configuring, and customizing Globus is not a trivial task, and currently requires a highly skilled support team. Therefore, current Grid solutions make excellent sense for dozens of large labs that work together on similar problems. Again, that is not the case for most labs around the world.

The vast majority of research labs are small (a dozen people or so), focus their research on some narrow topic (as to have a chance to compete in the “scientific ecosystem”), do not belong to top Universities (as by definition top places are few), and cannot count on having a cutting-edge computer support team (due to all above). Yet, these labs increasingly demand large amounts of computational power, just as large labs and high-visibility projects do.

This paper provides a comprehensive description of OurGrid, a system designed to fill this gap, catering for small and medium-sized labs around the world that have unserved computational demands. *OurGrid is an open, free-to-join, cooperative Grid in which labs donate their idle computational resources in exchange for accessing other labs’ idle resources when needed*. It uses the Network of Favors, a peer-to-peer mechanism that makes it in each lab’s best interest to collaborate with the system by donating its idle resources. OurGrid leverages the fact that people do not use their computers all the time. Even when actively using computers as research tools, researchers alternate between job execution (when they require computational power) and result analysis (when their computational resources go mostly idle). In fact, in most small and medium labs, people have relatively small jobs, for which they seek fast turnaround times, so as to speed-up the run/analyze iterative cycle. Interestingly, such behavior has also been observed among users of Enterprise Desktop Grids [41].

For OurGrid to succeed, it must be *fast*, *simple*, *scalable*, and *secure*. Clearly,OurGrid must be *fast*, i.e. the turnaround time of a job must be much better than that which is possible using only local resources. In particular, the user is not interested in some system-wide metric such as throughput [41]. The user must see her own application running faster; otherwise she will see no value in OurGrid and will thus abandon the system. *Simplicity* is also a fundamental requirement for OurGrid. After all,

labs want to spend the minimum possible effort on the computer technology that will solve their problems. They want to focus on the research that they do. Computers are just tools for them. Another key requirement for OurGrid is *scalability*. OurGrid must scale well; otherwise it will not tap the huge amount of computational power that goes idle in the labs around the world. Note that scalability is not just a technical issue. It also has administrative implications. In particular, it is not acceptable to have to go through a human negotiation to define who can access what, when and how (something that is needed to set up current Grids). OurGrid must be a free-to-join open Grid. Finally, OurGrid must be *secure*, because its peer-to-peer automatic granting of access will allow unknown foreign code to access one's machine. Nevertheless one's machine must remain safe.

Achieving these goals is a very challenging task, which gets even tougher because (i) current Internet connectivity is far from universal (due to firewalls and private addressing), and (ii) new vulnerabilities appear on a daily basis. In order to simplify the problem somewhat, at least for now, we reduce OurGrid's scope to supporting Bag-of-Tasks (BoT) applications. BoT applications are those parallel applications whose tasks are independent. Despite their simplicity, BoT applications are used in a variety of scenarios, including data mining, massive searches (such as key breaking), parameter sweeps, simulations, fractal calculations, computational biology, and computer imaging. Assuming applications to be BoT simplifies our requirements in a few important ways. In particular, since a failed task does not affect other tasks, we can deliver fast execution of applications without demanding any QoS guarantees from the resources. It also makes it easier to provide a secure environment, since network access is not necessary during the execution of a foreign task.

In short, by focusing on BoT applications, we can deliver a useful (though not complete) solution for the compute-hungry labs of the world, now. In fact, OurGrid is in production since December 2004. Any lab can join the system by downloading the code from http://www.ourgrid.org. Note that no human contacts or negotiation are needed for a new lab to join the system. At the time of the writing of this paper, OurGrid comprises about a dozen labs and a few hundred machines. The current state of the system is available at http://status.ourgrid.org. Naturally, OurGrid is open source.

The rest of this paper is organized as follows. Section 2 defines OurGrid's scope and presents its architecture. Section 3 describes how we make it in each lab's best interest to contribute resources to the Grid. Section 4 explains how we use Xen virtual machines [8] to deal with the security concerns inherent to an open free-to-join Grid. Section 5 describes how OurGrid is used, including how the Grid heterogeneity is hidden from the user, and how scheduling promotes the application performance. Section 6 discusses the experiences gained in developing OurGrid, and Section 7 discusses the process of putting the system in production. Section 8 compares OurGrid with related work. Finally, Section 9 closes the paper with our final remarks.

2. Scope and Architecture

OurGrid's goal is to enhance the computing capabilities of research labs around the world. For now, at least, OurGrid assumes applications to be Bag-of-Tasks (BoT). However, a single OurGrid task may itself be a parallel tightly coupled application (written in MPI, for example). Although OurGrid does not run a parallel task across the Grid, it may very well run it on a remote site. OurGrid can use both interactive desktop computers and dedicated clusters (which may be controlled by a resource manager, such as Maui, OpenPBS, and LSF).

OurGrid strives to be non-intrusive, in the sense that *a local user always has priority access to local resources*. In fact, the submission of a local job kills any foreign jobs that are running locally. This rule assures that OurGrid cannot worsen local performance, a property that has long been identified as key for the success of resource-harvesting systems [60].

OurGrid is designed to be scalable, both in the sense that it can support thousands of labs, and that joining the system is straightforward. For scalability, OurGrid is based on a peer-to-peer network, with each research lab corresponding to a peer in the system. However, peer-to-peer sys-

tems may have their performance compromised by freeriding peers [38, 50, 55]. A *freerider* is a peer that only consumes resources, never contributing back to the community. This behavior can be expected to have a very negative impact on OurGrid, because many users reportedly have an 'insatiable demand for computer resources', and thus we do not anticipate having a resource surplus to give to freeriders. We have dealt with this problem by creating the Network of Favors, a totally decentralized and autonomous allocation mechanism that marginalizes freeriders. Section 3 describes the Network of Favors.

A given lab in OurGrid will commonly run tasks from other unknown labs that are also part of the community. This creates a very obvious security threat, especially in these days of so many software vulnerabilities. Therefore, we must provide a way to protect local resources from foreign unknown code. That is the job of SWAN (Sandboxing Without A Name), a solution based on the Xen virtual machine [8], which isolates the foreign code into a sandbox, where it can neither access local data nor use the network. Naturally, we must also protect the application from malicious labs. This can be done with low overhead by using the credibility-based sabotage detection proposed by Sarmenta [54]. Section 4 describes OurGrid's security.

Users interact with OurGrid via MyGrid, a personal broker that performs application scheduling and provides a set of abstractions that hide the Grid heterogeneity from the user. The great challenge in scheduling is how to assure good performance for the application in a system as large and loose-coupled as OurGrid. In particular, the scale of the system makes it hard to obtain reliable forecasts about the execution time of a given task on a given processor. To deal with this problem, we have devised schedulers that use task replication to achieve good performance in the absence of performance forecasts. As for the abstractions, the goal is to balance between ease-of-use and performance, while considering the limitations of

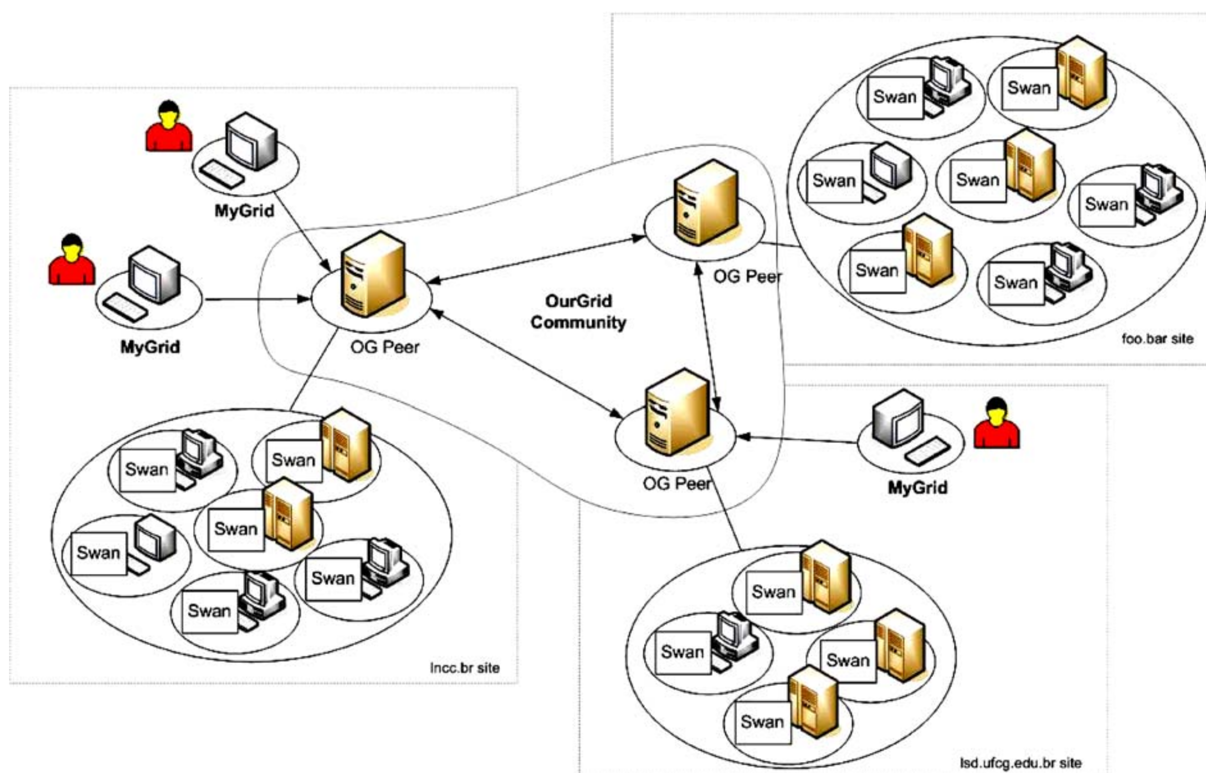


Figure 1 OurGrid architecture.

the current lack of general connectivity in the Internet [59]. Section 5 presents MyGrid.

In summary, OurGrid has three main components: The OurGrid peer, the MyGrid broker, and the SWAN security service. Figure 1 shows them all, depicting the OurGrid architecture.

3. Promoting Cooperation

OurGrid is based on a peer-to-peer resource-sharing network through which labs with idle resources donate them to labs with computational demands. A potential problem for any resource-sharing network is that some users may *freeride*, that is, they may consume resources offered by others but offer none of their own. Freeriding can be achieved by simply creating a peer with no resources to offer, or (slightly less simply) by hacking the OurGrid code. Experience with file-sharing peer-to-peer systems shows that in the absence of incentives for collaboration, a large proportion of the peers only consume the resources of the system [38, 50, 55]. This problem is especially pertinent for a network such as OurGrid which connects unknown sites, and which has an expected high demand for the resource being shared (namely, computational power). Freeriding is a serious concern because the more labs freeride, the smaller the system utility becomes, potentially to the point of system collapse.

One way to address this problem is to implement a market-based resource allocation mechanism [1, 13, 14, 47]: Users are required to pay the resource owners for the use of their resources, either in real currency or in an artificial currency only valid within the network. To make this solution work across organizational boundaries it is necessary to have secure and reliable global accounting and billing mechanisms. Implementing such mechanisms is challenging, and we believe that the difficulty of doing this has been a major factor limiting the wide adoption of multi-organizational Grids.

Accounting and billing systems may be necessary in any case when resources are shared between different commercial organizations. However, for non-commercial organizations, in particular labs carrying out eScience research, there is room for a different and more lightweight way to promote cooperation. In fact, OurGrid uses the Network of Favors to discourage freeriding *without requiring a billing system*, and employs an autonomous accounting mechanism that solely uses information local to the peer, therefore *without requiring a Grid-wide accounting system*. The two mechanisms together make OurGrid resistant to freeriding and yet lightweight and easy to deploy.

3.1. Network of Favors

OurGrid provides an incentive for peers to cooperate, by using a peer-to-peer reciprocation scheme called the *Network of Favors*. A key criterion for the design of this scheme was that it should be particularly lightweight and easy to implement in real systems. The Network of Favors has been described and evaluated in [4–6]. We give an overview of it here.

In the Network of Favors, a favor is the allocation of a processor to a peer that requests it, and the value of that favor is the value of the work done for the requesting peer. Each peer A keeps a local record of the total value of the favors it has given to and received from each peer B with which it has interacted, denoted, respectively, by $v_A(A,B)$ and $v_A(B,A)$. (We shall soon see how the autonomous accounting mechanism is used to determine $v_A(A,B)$ and $v_A(B,A)$).

If A has an idle processor that is requested by more than one peer, A calculates a local ranking value $R_A(B)$ for each requesting peer B based on these numbers, using the function

$$R_A(B) = \max\{0, v_A(B, A) - v_A(A, B)\}$$

and donates the use of the processor to the requesting peer with the highest ranking. The rationale is that each peer autonomously prioritizes donations to the peers to whom they owe most favors, motivating cooperation. Note that the ranking value is used *only* to prioritize a peer B, to which the donating peer A owns more favors, over a peer C, to which peer A owns less favors. Resource donation always happens. In particular, a peer C with $R_A(C) = 0$ (whether a newcomer who has not yet given any favors, or a peer who

has temporarily received more favors from A than it has given to A) can still receive a favor from A, provided that no peer B with $R_A(B) > 0$ requests the same resource at the same time.

A key feature of our ranking scheme is that each peer performs it in a completely autonomous way. The value $R_A(B)$ will be based solely on A's past interactions with B, and in general will not be equal to $R_C(B)$, where C is a third peer. No attempt is made to combine or reconcile these values to determine a global ranking value for B. There is therefore no requirement for the labs involved in OurGrid to adopt special mechanisms to ensure the integrity of information received from peers about their interactions with third parties, such as the shared cryptographic infrastructures or specialized storage infrastructures used by some peer-to-peer reputation schemes [24, 40]. This allows our ranking scheme to be very lightweight. Furthermore, there is little scope for malicious peers to distort the rankings: Since $R_\mathrm{A}(B)$ is based only on interactions directly involving A and B, strategies based on lying about the behavior of third parties cannot be applied, and since the local ranking for a peer is always non-negative and is zero if the peer is new to the system, a malicious peer cannot increase its ranking by leaving the system and re-entering as a newcomer.

Naturally, for this scheme to work, peers must be able to find each other in the first place, and we need to protect the system against impersonation of one peer by another. We currently use a simply centralized discovery mechanism on which peers register themselves and discover the other peers in the system. Peers thus form a clique within which a request is directly sent from the originating peer to all other peers. While this simple scheme causes no problem at the current system scale (see Section 7), it clearly poises serious scalability limitations. Consequently, we are currently replacing it by NodeWiz, a fully decentralized peer-to-peer request propagation system [9]. Impersonation of peers with high reputation can be prevented by a simple use of asymmetric cryptography, in which a peer autonomously selects a public key to identify itself to other peers. Note that this does not require a shared system-wide public key infrastructure.

We have demonstrated through simulations and analytical modeling that the Network of Favors is indeed effective at discouraging freeriding, provided that there is enough contention for resources [4–6]. Figure 2 exemplifies our findings. It shows the results of three simulations of 100 peers, $100 \cdot f$ of whom are freeriders who request all available processors at each turn, and the rest of

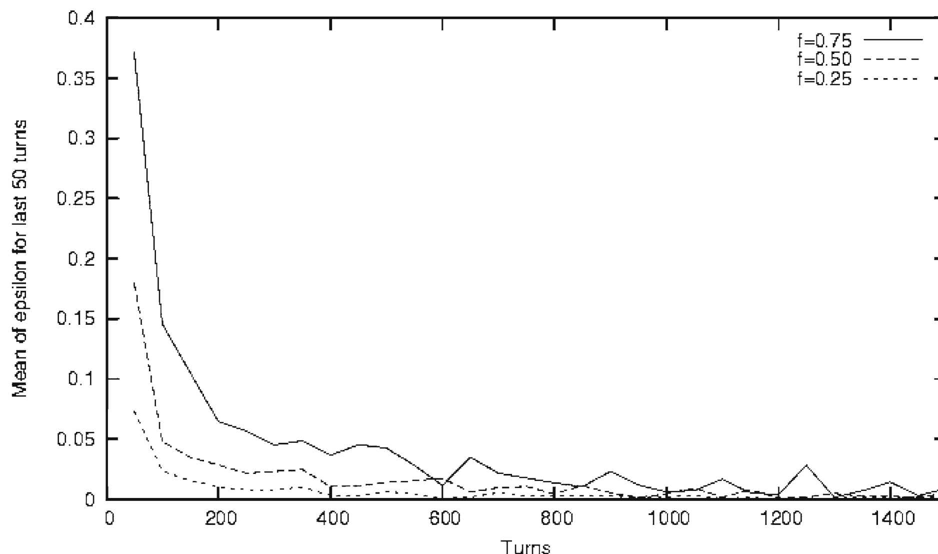


Figure 2 Resources freeriders manage to obtain.

whom with probability 1/2 at each turn either offer a compute cycle to the community or request it. Accounting was perfect, i.e. the values of v_A(A,B)and v_A(B,A) were provided by an omniscient oracle. Figure 2 plots epsilon, the fraction of the community's resources consumed by freeriders. As can be seen, epsilon becomes very small as the system reaches steady state. Therefore, it becomes in the best interest of each peer not to freeride.

3.2. Autonomous Accounting

For the Network of Favors to work, each peer A must be able to determine $v_A(A,B)$ and $v_A(B,A)$ for each peer B with whom it interacted. In order to keep the system easy to deploy, we want to avoid having a Grid-wide service that everybody must trust. Therefore, peer A must be able to *autonomously* determine $v_A(A,B)$ and $v_A(B,A)$.

A very simple option is to use time to determine $v_A(A,B)$ and $v_A(B,A)$, setting $v_A(A,B)$ equal to $t(A,B)$, the length of time that A has made a machine available to B. $v_A(B,A) = t(B,A)$ would be how long A has used a machine made available by B. Clearly, A can compute both values autonomously, without demanding any information from other peers.

The problem with the time-based accounting is that it gives a peer A an incentive to run multiple tasks simultaneously on each of its processors. This strategy gives two advantages to A. First, it enables A to run many more tasks, thus providing more favors to more peers. Second, it increases the execution time of each task, which is a good thing for A, because other peers value the favors of A based on how long A took to run their tasks. This may also decrease the power of the Grid as a whole, since it introduces thrashing when all tasks that share a machine cannot fit in main memory. As tasks take longer to finish, the probability of task preemption also increases.

We deal with this problem by using a simple relative accounting scheme. This scheme assumes that all jobs have some of their tasks executed locally. Since local jobs preempt remote jobs in OurGrid, this assumption seems quite reasonable. We thus set $v_A(B,A) = t(B,A) \times \mathrm{RP}_A(B)$, where $\mathrm{RP}_A(B)$ denotes the relative power of B, compared to A, i.e. $\mathrm{RP}_A(B) = e(A) / e(B)$, where $e(A)$ is the average execution time of A's tasks running locally, and $e(B)$ is the average execution time of A's tasks running at B. We also set $v_A(A,B) = t(A,B)$. (That is, $\mathrm{RP}_A(A) = 1$).

How well this scheme works will naturally depend on the Grid heterogeneity (how machines differ) as well as on the job heterogeneity (how tasks differ in terms of their computational demands). For the vast majority of scenarios we investigated, however, this scheme performs very

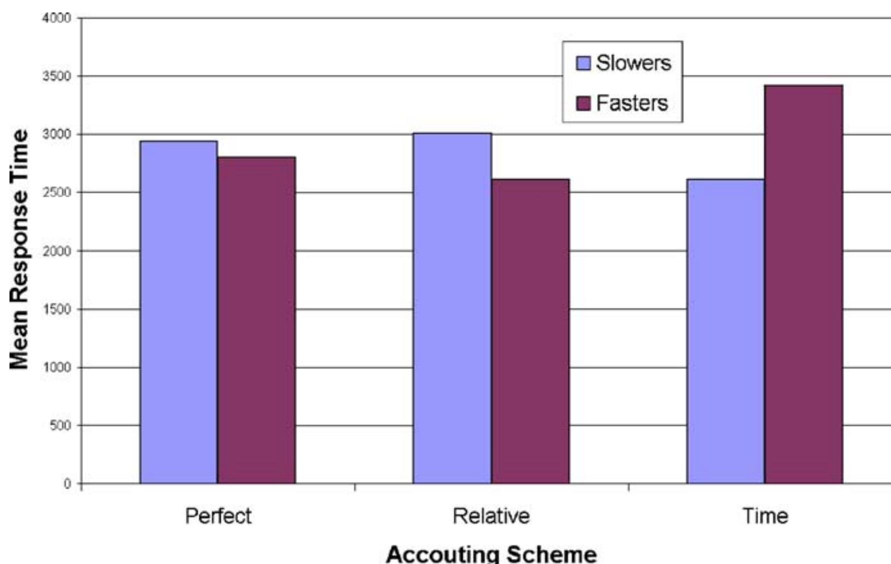


Figure 3 Mean response time of jobs using perfect, time, and relative accounting.

well [52]. Figure 3 gives a good example of our results. It shows the mean response time of jobs ran in a Grid with 20 peers, each with 25 machines whose relative speed varies following the uniform distribution U(4,16). However, 10 of such peers run two tasks on each machine, appearing for the Grid to have 50 machines of speed U(2,8). We call these peers *slowers*, whereas the other 10 peers (which run one task per machine) are called *fasters*. Jobs arrive every U(1,799) time units, each consisting of 250 tasks, each task requiring U(200,600) time units to execute on a machine with relative speed equal to one. Note that the workload submitted to each peer is enough to saturate the peer, but due to the probabilistic job arrival times, the peer is nevertheless sometimes idle. The simulations were run using three accounting schemes, the aforementioned *time* and *relative* schemes, and the *perfect* scheme, in which an omniscient oracle reveals exactly how much compute power each task consumed. As Figure 3 shows, using perfect accounting, the Network of Favors is almost insensitive to whether peers are faster or slower. (The small difference is due to the greater chance of preemption of a remote task by a local task in the slower peers.) As expected, time-based accounting gives an advantage to slower peers, creating an incentive for peers to adopt this strategy. The proposed relative autonomous accounting gives a small advantage to the faster peers, compared to the perfect scheme. However, this advantage is small (around 10% in this scenario). More importantly it discourages the machine-sharing strategy, and therefore avoids a decrease in the overall utility of the system.

4. Dealing with Security

The Network of Favors makes it feasible to create a free-to-join Grid in which labs not necessarily know each other, yet have an incentive to collaborate with the community. While this greatly aids system growth, it raises serious security concerns. First, how can we protect an application from a malicious lab? Second, how can we protect a lab from a malicious application?

Note that the Network of Favors creates an incentive for a lab to sabotage tasks, meaning that tasks are not executed and bogus results are returned instead. By pretending to have executed tasks, a saboteur lab could rapidly put other labs in debt to it. We intend to deal with this issue by judiciously replicating tasks using the credibility-based sabotage detection scheme proposed by Sarmenta [54]. Currently, we allow the application to test the results processed in the Grid using application-specific checks. Another issue regarding protecting an application from a malicious lab is *data privacy*, i.e., how we can ensure that a lab does not read the data of a guest task. Although this is an important concern, we do not provide any mechanism to handle this potential problem. We assume that applications are not very secretive, thus requiring no data security, which seems to be an appropriate assumption for most research labs. Moreover, the application data is naturally scattered throughout the Grid, making it much harder for a few malicious labs to obtain much of the application data.

We address the converse issue by proposing SWAN (Sand-boxing Without A Name), an execution environment that protects the provider's resource against malicious or buggy foreign task. From the task standpoint, SWAN is totally transparent, i.e., the task does not need to be modified in order to run on SWAN. SWAN builds on other research efforts that try to ensure security and integrity of shared resources in a distributed environment. These solutions can be classified by the kind of isolation used: Enhanced access control [45], system call interposition [26, 49], and virtual machines (VM) [31, 36].

Enhanced access control and system call interposition are very similar. They try to improve security at the operating system level since available operating systems do not efficiently isolate sensitive resources from applications, basing the access control only on user identity and resource ownership [44]. These approaches define a specific security policy for each application. They vary on how they implement the security policy, with enhanced access control demanding changes in the operating system kernel.

In the VM approach, the operating system runs on top of a hypervisor that creates an isolated environment and controls all physical resources. Application software inside the virtual machine

can only access resources that have been explicitly allocated to its host operating system by the hypervisor.

We evaluated these three mechanisms according to security, performance and intrusiveness, and came to the conclusion that the VM approach is the best choice for OurGrid. Security was the decisive aspect in our choice. VM offers the best isolation because it offers 'double barrier' isolation to an attacker. Even when an attacker trespasses the first barrier (the native operating system security mechanism) and gains unauthorized access to resources, he is contained inside a virtual machine. Furthermore, systems based on system call interposition or enhanced access control are hard to configure and maintain [35]. The configuration may not cover all security breaches present in the operating system.

Xen is a virtual machine implementation that achieves excellent performance, especially for CPU-intensive applications [8]. However, it does require changes in the operating system kernel. Despite its intrusiveness, we opted for Xen as the basis of our security infrastructure due to its better security and performance. SWAN goes one step further than Xen. Because the applications are BoT, i.e., no communication is required during the execution, SWAN disables the virtual machine network access. This is essential to prevent a malicious task behind the lab's firewall from exploring vulnerabilities in the surrounding machines. In fact, when remotely executing a task on a given computer, OurGrid requires a network connection for transferring data, launching, monitoring, and retrieving the results from the task. But the task itself does not need to access the network. The idea behind SWAN is to enforce security by creating two distinct security levels for execution, referred to as trusted and non-trusted, based on the resource requirements of OurGrid and the BoT applications.

As shown in Figure 4, the trusted level runs OurGrid software and has access to the network. OurGrid software is assumed to be secure for two reasons. First, it does not change from the execution of one task to another. Second, it is open source and the lab's system administrator can check it. At the non-trusted level, only the basic resources necessary for executing the remote tasks are available. Since these are BoT applications, they do not require network access, and this is not provided. Furthermore, the non-trusted level has a file system that is completely isolated from the trusted level.

SWAN also implements another security mechanism, the *sanity check*, which uses checksums to verify the integrity of the non-trusted VM after each task execution. This feature prevents

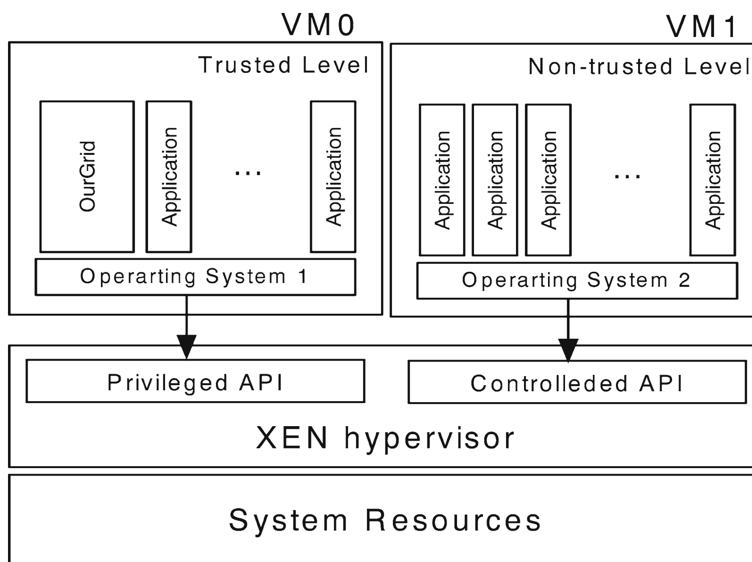


Figure 4 SWAN architecture.

malicious tasks from installing a Trojan horse or modifying the virtual machine file system in order to attack the next task running on the machine.

5. Making it Simple and Fast

Now that we have a peer-to-peer resource-sharing scheme that provides sensible incentives and a sandboxing mechanism that addresses the security worries that come with the use of such a scheme, we must provide the user with a convenient way to use the system. First, we must hide the Grid heterogeneity from the user. After all, in a free-to-join Grid such as OurGrid, the user will often face a totally unknown machine, with unknown hardware architecture, operating system, installed software, and file system organization. Second, we must ensure that the user's application runs quickly. Since many users see OurGrid as a way to speed up the run/analyze cycle for computer-based research, we must assure fast application turnaround, not just system-wide asymptotic throughput. In OurGrid, these functionalities are provided by MyGrid, a personal broker through which the user interacts with the Grid, as depicted in Figure 1. We will now describe how MyGrid provides such functionalities. For the internal architecture and deeper details of MyGrid, we refer the reader to [19, 48, 53].

5.1. Hiding Heterogeneity

Recall that OurGrid deals with BoT applications, i.e. applications comprised of independent tasks. A task is formed by *initial*, *Grid*, and *final* subtasks, which are executed sequentially in this order. Subtasks are external commands invoked by OurGrid; consequently, any program, written in any language, can be a subtask. The initial and final subtasks are executed locally, on the user machine. The initial subtask is meant to set up the task's environment by, for instance, transferring the input data to the Grid machine selected to run the task. The final subtask is typically used to collect the task's results back to the user machine. The Grid subtask runs on a Grid machine and performs the computation per se. In addition to its subtasks, a task definition also includes the *Grid machine requirements*, as we shall shortly see.

MyGrid abstractions allow users to write subtasks without knowing details about the Grid machines on which they will be run, such as how their file systems are organized. The abstractions *storage* and *playpen* represent storage spaces in the unknown Grid machine, which the task refers to as $STORAGE and $PLAYPEN, respectively. File transfer commands are available to send and retrieve files in the storage spaces.

Storage is a best-effort persistent area: Files stored there *may* be available for the next task to run on the machine. Storage is useful for distributing files that are going to be used more than once, such as program binaries. Since storage does not guarantee the presence of a file previously transferred, the task description must always state which files are to be sent to the storage area. Unnecessary transfers are avoided by checking the existence, the modification date and a file hash before sending the file. Playpen provides temporary storage, serving as the working directory for the Grid subtask, and disappearing when the task terminates. The Grid machine's owner can specify the directories that are going to be presented as storage and playpen, as well as the maximum size they can reach. Storage space is managed using the least-recently used policy. We recommend for storage to be configured in a file system that is mounted by all machines in the lab. By doing so, transferring a file to storage makes it available to all machines within that lab.

In order to make it easier to write the initial and final subtasks, MyGrid also defines the environment variables $PROC, $JOB, and $TASK. They, respectively, contain the Grid machine chosen to run the task, the job unique number, and the task unique number (within a job). For example, suppose the user wants to run the binary task, which has the file IN as input and the file OUT as output. The initial subtask would then be:

```
store task $STORAGE
put IN $PLAYPEN
```

The Grid subtask would be simply:

```
$STORAGE/task < IN > OUT
```

And the final subtask would collect OUT to the results directory, renaming the file by appending the unique job/task number to its name.

```
get $PLAYPEN/OUT
results/OUT-$JOB-$TASK
```

Appending the job and task numbers to OUT ensures the uniqueness of each output in the quite common cases where a given application is run many times, and/or several tasks produce output with the same name.

The final component of a task is its *Grid machine requirements*. In OurGrid, Grid machines are labeled with *attributes*. Attributes have values and simple Boolean expressions can be used to specify the needed machine characteristics. Predefined attributes include playpensize, storagesize, opsys, and arch. For example, when a task requires opsys = linux and arch = IA32, it demands Intel 32-bit compatible machines running Linux.

Any subtask can also use the attributes of a Grid machine. This makes it possible for subtasks to adapt to different kinds of Grid machines. Refining the above example, suppose that task has binaries for Linux and Windows, placed, respectively, at linux and windows directories. The initial subtask could then use the opsys attribute to mirror the correct binary, as shown below.

```
if (opsys = linux) then
    store linux/task.sh $STORAGE/task
else
    store windows/task.bat $STORAGE/task
endif
put IN $PLAYPEN
```

It is interesting to contrast OurGrid's approach to hiding Grid heterogeneity with the approach used by Condor. Condor hides the differences among Grid machines by intercepting the system calls and forwarding them to the user machine [60]. This way, a task running in a remote machine has the illusion that it is running locally. We see these two approaches as representing different trade-offs in the design space of Grid working environments. Condor's working environment approach is simpler for the user to understand, but requires re-linking the application with Condor's redirection library. OurGrid's working environment approach is not complicated, but it does require the user to learn a few new concepts. On the other hand, OurGrid's approach does not require end-to-end communication between the user machine and the Grid machine, simplifying the deployment of OurGrid. Which approach has the best performance is heavily dependent on the application. If an OurGrid application transfers a large file but only uses part of it, performance would be better with Condor. If a Condor application accesses the same small file many times, OurGrid is likely to do better. Finally, Condor's approach implies that a write is immediately propagated to the user machine, whereas in OurGrid results go to the user's machine only when tasks finish. This makes fault recovery easier in OurGrid (simply resubmit the task, without worrying about side effects) and enables task replication, an important issue for efficient scheduling as discussed next.

5.2. Promoting Application Performance

Despite the simplicity of BoT applications, scheduling BoT applications on Grids is difficult due to two issues. First, efficient schedulers depend on information about application (such as estimated execution time) and resources (processor speed, network topology, load, and so on). However, it is difficult to obtain accurate information in a system as large and widely dispersed as a Grid. Second, since many important BoT applications are also data-intensive applications, considering data transfers is paramount to achieve good performance. Thus, in order to achieve efficient schedules, one must provide coordinated scheduling of data and computation.

MyGrid's first scheduler (Workqueue with Replication or simply WQR) dealt only with the first issue. WQR uses no information about tasks or machines. It randomly sends a task to a machine; when a machine finishes, another task is sent to it; when there are no more tasks to send, a randomly chosen running task is replicated. Replication is the key to recovering from bad allocations of tasks to machines (which are inevitable, since WQR uses no information). WQR performance is as good as traditional knowledge-based schedulers

fed with perfect information [48]. It did consume more cycles though. However, this was noticeable only when the number of tasks was of the same order of magnitude as the number of machines, or less [48]. And, somewhat surprisingly, limiting replication to 2× (i.e. the original and the replica) delivered most of the performance improvement, while resource waste was limited to around 40% for the extreme case with three machines per task [48]. However, WQR does not take data transfers into account.

With version 2.0 of MyGrid, we released an alternative scheduler for MyGrid, called Storage Affinity [53], which does tackle both problems simultaneously. (WQR is still available within MyGrid because it does quite a good job with CPU-intensive BoT applications.) There are a few Grid schedulers that take data transfers into account in order to improve the performance of the applications. However, all these schedulers require a priori knowledge of the execution time of the tasks on each processor that composes the Grid, and this varies dynamically with the Grid load. Storage Affinity, on the other hand, does not use dynamic, hard-to-obtain information. The idea is to exploit data reutilization to avoid unnecessary data transfers. The data reutilization appears in two basic flavors: *Inter-job* and *inter-task*. The former arises when a job uses the data already used by (or produced by) a job that executed previously, while the latter appears in applications whose tasks share the same input data.

We gauge data reutilization by the *storage affinity* metric. This metric determines *how close* to a site a given task is. By *how close* we mean *how many bytes* of the task input dataset are already stored at a specific site. Thus, the *storage affinity* of a task to a site is the number of bytes within the task input dataset that are already stored in the site. Information on data size and data location can be obtained a priori with less difficulty and loss of accuracy than, for example, information on CPU and network loads or the completion time of tasks. For instance, information on data size and location can be obtained if a data server at a particular site is able to answer requests about *which* data elements it stores and *how large* each data element is. Alternatively, an implementation of a Storage Affinity heuristic can easily store a history of previous data transfer operations containing the required information.

Naturally, since Storage Affinity does not use dynamic information about the Grid and the application, inefficient task-to-processor assignments do occur. In order to circumvent this problem, Storage Affinity uses a task replication strategy similar to that used by WQR [48]. Replicas have a chance of being submitted to faster processors than those processors assigned to the original task, thus decreasing average task completion time.

To test Storage Affinity, we performed a total of 3,000 simulations in a variety of scenarios [53]. The Grid and the application varied in heterogeneity across the scenarios. The application was assumed to have a 2GB input, and reuse was inter-job. The simulations compared the efficiency of Storage Affinity against XSufferage and WQR. XSufferage [16] is a famous scheduler that takes data placement into account, but requires knowledge of the execution time of the tasks on each processor that composes the Grid. WQR was considered for the opposite reason: It does not need information, but does not consider data placement either. Each simulation consisted of a sequence of three executions of the same job in a given Grid scenario. These three executions differed only in the scheduler used (WQR, XSufferage and Storage Affinity).

Table 1 presents a summary of the simulation results. As can be seen, on average, Storage Affinity and XSufferage achieve comparable performances. The results show that both data-aware heuristics attain much better performance than WQR. This is because data transfer delays dominate the execution time of the application,

Table 1 Storage affinity results

		Storage affinity	WQR	XSufferage
Execution time (s)	Mean	14,377	42,919	14,665
	Std dev	10,653	24,542	11,451
Wasted CPU (%)	Mean	59.24	1.08	N/A
	Std dev	52.71	4.12	N/A
Wasted bandwidth (%)	Mean	3.19	130.88	N/A
	Std dev	8.57	135.82	N/A

thus not taking them into account severely hurts the performance of the application. In the case of WQR, the execution of each task is always preceded by a costly data transfer operation (as can be inferred from the large bandwidth and small CPU wastage). This impedes any improvement that the replication strategy of WQR could bring. On the other hand, the replication strategy of Storage Affinity is able to cope with the lack of dynamic information and yields a performance very similar to that of XSufferage. The main inconvenience of XSufferage is the need for knowledge about dynamic information, whereas the drawback of Storage Affinity is the consumption of extra resources due to its replication strategy (an average of 59% extra CPU cycles, and a negligible amount of extra bandwidth). However, as with WQR, the resources wasted by Storage Affinity can be controlled by limiting replication, with very little impact on performance [53].

Note that, to avoid scalability limitations, OurGrid has no single 'superscheduler' that oversees all scheduling activity in the system. Each user runs a MyGrid broker, which schedules the user's jobs and competes with other MyGrid brokers for the system's resources. However, since MyGrid's schedulers use task replication, some resources will be wasted by executing replicas, instead of executing someone else's task. In fact, this extra consumption of resources raises important concerns about the system-wide performance of a distributed system with multiple, competing replication schedulers [41]. We are currently investigating this issue. Our preliminary results suggest that performance degradation is low for the case in which each scheduler deals with many more tasks than the resources available for scheduling. Alas, the case there are more processors than tasks is troublesome. However, a simple referee strategy (which can be locally and autonomously implemented by each resource) greatly improves matters. The local referee strategy consists of having resources restrict themselves to serving one scheduler at a time. This strategy improves the emergent behavior of a system with competing replication schedulers in all scenarios we have investigated so far, and appears to be critical in ensuring good system performance when there are more resources than tasks.

6. Implementing the Vision

While we were designing and analyzing OurGrid's main components, we were also implementing them with the goal of producing a production-quality open-source system. OurGrid is written in Java and supports the execution of Linux and Windows BoT applications. OurGrid's current version is 3.2, which can be downloaded from http://www.ourgrid.org under GPL.

Reality is always more complex than our models, and thus an enormous amount of effort had to be put in generating software that can reliably work in production, outside our own lab. In peak periods, we had more than 20 people working on OurGrid, including undergraduates, graduate students, and staff. This section is meant to highlight what we consider the most interesting aspects of this experience.

6.1. Isolation Interfaces

In order to assure modularity and easy interoperability with other technologies, we designed OurGrid around two main isolation interfaces: GridMachine and GridMachineProvider. GridMachine (or GuM) represents a processor machine in the system. It exports methods to remotely execute a command, and to transfer files to and from the processor. We have three implementations of GuM. The first invokes user-defined scripts for those tasks. It enables the use of ssh and scp as 'Grid middleware', thus eliminating the need for Grid deployment on the worker machines. The second implementation uses GRAM and GridFTP to provide access to Globus machines. The third is OurGrid's native Java-based implementation. It provides faster access and better instrumentation than the other implementations. GuM implementations can run under SWAN to isolate malicious or erroneous foreign code.

A GridMachineProvider (or GuMP) is a source of GuMs. It supports a very simple wannaGuM primitive, which is eventually responded to with hereIsGuM. GuMPs abstract the fact that the machines that compose the Grid are under control of multiple entities. For example, machines may be controlled by queue managers such as PBS,

Maui, and LSF. Idle interactive machines may be under the control of an OurGrid peer itself, or be managed as a Condor pool. All these sources of GuMs are hidden behind the very simple GuMP interface. A GuMP can also funnel all communications between its GuMs and the external world, providing a reasonable way to deal with private IPs and firewalls. Note also that GuMPs can trade GuMs among themselves by using the Network of Favors (see Section 3).

6.2. Dealing with Space-shared Machines

Many labs with large computational demands have access to space-shared machines, composed of many processors, which do not share memory but are connected via high-speed networking. This is often in the form of a small, dedicated cluster, and sometimes in the form of remote access to a large supercomputer center. Space-shared resources are typically controlled by a queue manager, which is designed to promote the performance of parallel jobs. They are also a great source of compute power that we would like to use in OurGrid.

Enabling space-shared machines' idle processors to be accessed through the Grid is straightforward. Queue managers typically accept user-defined scripts that are invoked before and after a processor is allocated to a job. We use these scripts to start OurGrid's native implementation of GuM when a processor becomes idle, and to kill it when a processor is about to be allocated to a local job.

However, when a local user submits a job to OurGrid, it should be able to use the local space-shared resource as a local user, not only access its idle resources. This requires submitting a request to the queue manager that controls the resource. The request specifies (i) the number of processors needed, and (ii) the amount of time these processors are to be allocated to the job. The user can provide OurGrid with such information. However, we believe that the user just wants to run her application, and desires the least involvement possible with computer-related details. In particular, estimating execution time for a space-shared computer is a notoriously error-prone task [42]. Therefore, we devised heuristics to automatically craft space-shared requests on the user's behalf.

As a first solution, we implemented a simple static heuristic. The static heuristic simply issues the maximum number of allowed requests, each asking for the maximum number of processors and maximum amount of time allowed by the local policy. This greedy approach makes it possible to use space-shared resources without boring the user with space-shared requests, however it often produces large wait times (time spent by the requests waiting in queue before they can run).

In order to solve this problem, we devised an adaptive heuristic [22]. Its adaptation occurs over both requested time and number of processors. That is, the requests are dynamically crafted by learning from previous requests and the queue state of the resource, providing better throughput. The adaptive heuristic aims to maximize throughput. To calculate throughput, we must estimate task runtime. In the beginning, we have a default (constant) runtime estimate. If tasks could be successfully finished in this time, the estimated runtime will be the runtime of the longest task. If requested time is not enough to run tasks, the estimated task runtime will be the requested time multiplied by an integer factor. Our inspiration to enlarge or shrink the requested time is based on the TCP congestion window rationale: In bad situations make quick decisions, and in good ones be careful. Based on the estimated task runtime, the heuristic sweeps the request queue, choosing the best (greatest throughput) set of possible requests in a greedy manner. It is worth to point out that space-shared resources typically have a maximum number of pending requests per user (due to administrative policies). Thus, an initial set with a maximum number of pending requests allowed is created with the first possible requests. After that, if a new possible request could improve the throughput, a previous chosen request is discarded and the new request is inserted into the set of chosen requests. The chosen set is requested, and the process is repeated if the requests were not sufficient to run all tasks.

We evaluated this adaptive heuristic by simulating the arrival of a Grid job with two supercomputer workloads classically used for performance evaluation of space-shared resources: SDSC SP2 and CTC SP2 [30]. Table 2 summarizes the results. The speed-up denotes how much faster was the

Table 2 Performance of the adaptive heuristic to craft space-shared requests

Workload	Utilization (%)	Average speed-up
SDSC SP2	72	2.41
CTC SP2	55	14.81
CTC SP2 (×0.9)	61	1.98
CTC SP2 (×0.8)	68	1.77
CTC SP2 (×0.7)	78	1.84

adaptive heuristic compared with the static heuristic. As one can see, the results are quite good, especially for CTC SP2. We further investigated the cause of such difference [22] and found it to be due to the supercomputer utilization: CTC SP2 is considerably less loaded than SDSC SP2. In Table 2, we also show modified simulations of CTC SP2 on which the interarrival time was multiplied by 0.9, 0.8, and 0.7. As utilization increases due to the shorter interarrival time, the CTC SP2 speed-up becomes comparable to the SDSC SP2 speed-up. Nevertheless, even in these situations, the adaptive heuristic still runs approximately twice as fast as the static heuristic.

6.3. Software Engineering Grid Middleware

Developing production-quality software is much harder than developing a research prototype. We are doing research in order to determine what the software should look like, and hence have to deal with fuzzy requirements, which certainly complicate the software development. Furthermore, we have a large and very heterogeneous team, from undergraduates to fully qualified researchers. Finally, we must (i) assume as little as possible about the environment on which the software is installed, (ii) make the software as user-friendly as possible, and (iii) provide clear and comprehensive documentation. These items are necessary to increase user satisfaction and reduce support, but they too add to the challenge of producing production-quality software.

We had numerous problems with the quality of OurGrid in the first half of 2004. The software was buggy and unstable. The architecture was complex and not fully understood by the whole team. Bug fixes would frequently add new bugs. We here summarize how we addressed these problems. Readers interested in a detailed discussion of this experience should refer to [25].

To regain control of the software, we introduced a better defined software development process, heavily based on extreme programming (XP) [10], and changed the system internal architecture so as to simplify concurrent programming. We found XP to be a better fit to our environment because of its ability to deal with changing requirements, which are a commonplace in research. As preached by XP, we make intensive use of automated tests to reduce the chances that the evolution of the software introduces bugs. However, OurGrid is distributed and runs over asynchronous networks, and hence is non-deterministic, which makes writing automated tests for OurGrid quite challenging. We used AspectJ [7] to create some aspects that give some control over thread execution. In the tests of OurGrid macro-components (the peer, MyGrid, and the GuM implementations), other components are represented by mock objects, and the programmer states properties over all threads using AspectJ. Unfortunately, automated testing of the system as a whole remains an issue not satisfactorily solved.

Around May 2004, we went through a major refactor of the internal OurGrid architecture. The main goal was to improve modularity. We divided OurGrid's macro-components into modules that communicate asynchronously via events. For example, when MyGrid's Scheduler assigns a task's replica to a processor, it sends an event to the ReplicaExecutor asking for this assignment to be carried out. Each module reads events from a queue and processes them. This refactor greatly reduced the number of synchronized statements in the code (from 174 in OurGrid 2.1.3 to 110 in OurGrid 2.2, after the refactor), and, more importantly, isolated the modules. A thread in a module *never* goes into another module. This modularization enables us to reason about the multithreaded behavior of each module individually, greatly simplifying designing, coding, and debugging.

We are currently planning to extend this event-oriented architecture to the macro-components level, making OurGrid, MyGrid, and the GuM implementations communicate via events. For

that, we plan to replace Java RMI (the current communication mechanism used in OurGrid) by Jabber (an open standard-based instant messaging platform [39]). In fact, although RMI is great for client/server applications, it displays weaknesses when used by a distributed system such as OurGrid. First, it may be hard for the 'server' to call back a 'client', because a 'client' often is behind a firewall. Second, if the 'client' is blocked on a 'server' (because callbacks are hard), canceling such an invocation (for instance when we cancel a replica) requires special handling. Third, each pending RMI call blocks a thread, and thus the number of threads a Java Virtual Machine can effectively handle (a few hundred) quickly becomes a bottleneck. We hope to solve these issues with Jabber.

7. Reality-check of Deployment

As discussed above, getting quality software ready to be deployed can be a daunting task for a research lab. OurGrid's first version openly available for download dates back to mid-2002. However, it only contained MyGrid and two implementations of the GuM interface (native and user-defined scripts). This functionally allows the user to combine all machines she has access to (i.e. can log into) to create her personal Grid for running her BoT applications [19]. But it did not enable the creation of a worldwide free-to-join cooperative Grid, as described here. Due to the development problems in 2004, the OurGrid peer (and thus the ability to create the aforementioned cooperative Grid) was released only in October 2004, with version 3.0. After exhaustive testing by us and others, we declared OurGrid in production in December 2004, and it has been running ever since.

Note that OurGrid is a federation on which each peer keeps local autonomy. In fact, there is no single place that has a detailed Grid-wide view of the system. All we have is an indication of which peers are part of the Grid at a given moment. To obtain a Grid-wide view of OurGrid, we asked all peers for their local logs. Since peers delete old logs, we could get data from *all* peers only from March 5th 2005 on. After that, we automated

Figure 5 The evolution of the OurGrid Grid.

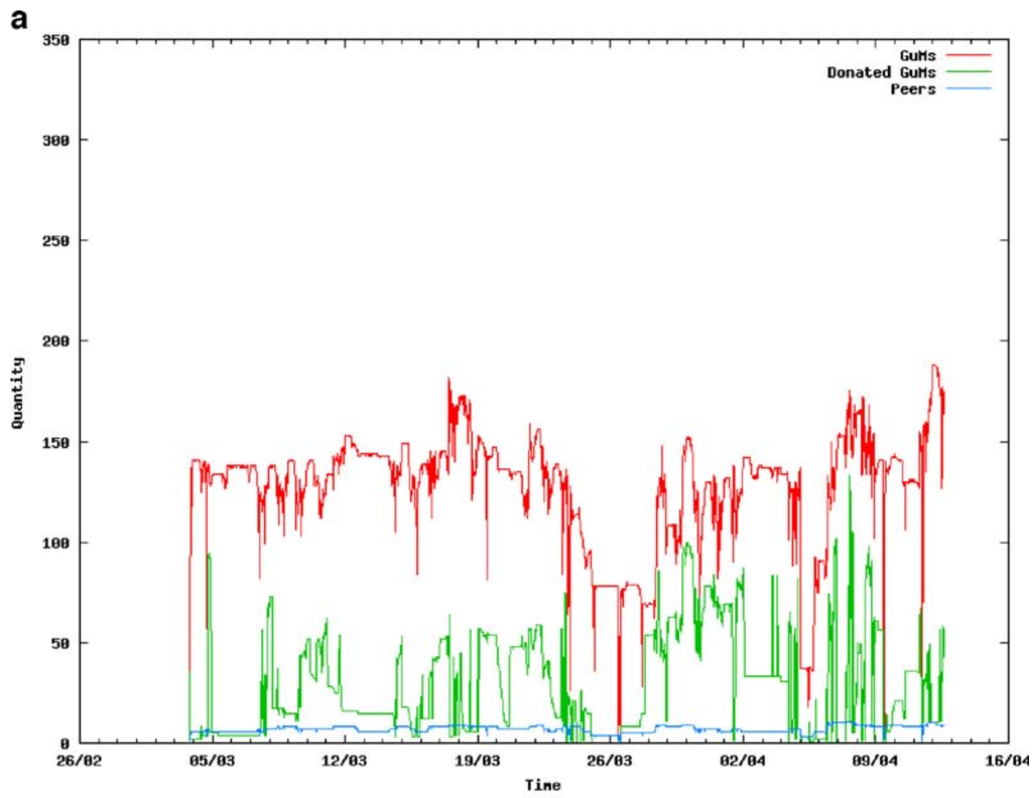
the process of log collection (with the peer owner approval, naturally). Unfortunately, however, a disk failure caused the loss of the logs from April 13th 2005 to July 28th 2005.

Figure 5 displays the available data on OurGrid (from March 5th to April 12th 2005 in Figure 5a and from July 29th 2005 to August 30th 2005 in Figure 5b). It plots the evolution of peers, machines, and machines donated to foreign peers. As expected, Grid conditions vary widely over time. Nevertheless, most of time there were machines being donated, implying that the Grid had labs with idle machines *and* labs with demand beyond their local resources. This supports our expectation that most labs have bursts of computational demand: They do not use their computational resources all the time; but when resources are needed, they can benefit from extra resources. Also, note that the number of peers and machines in OurGrid have grown considerably from March 2005 to July 2005.

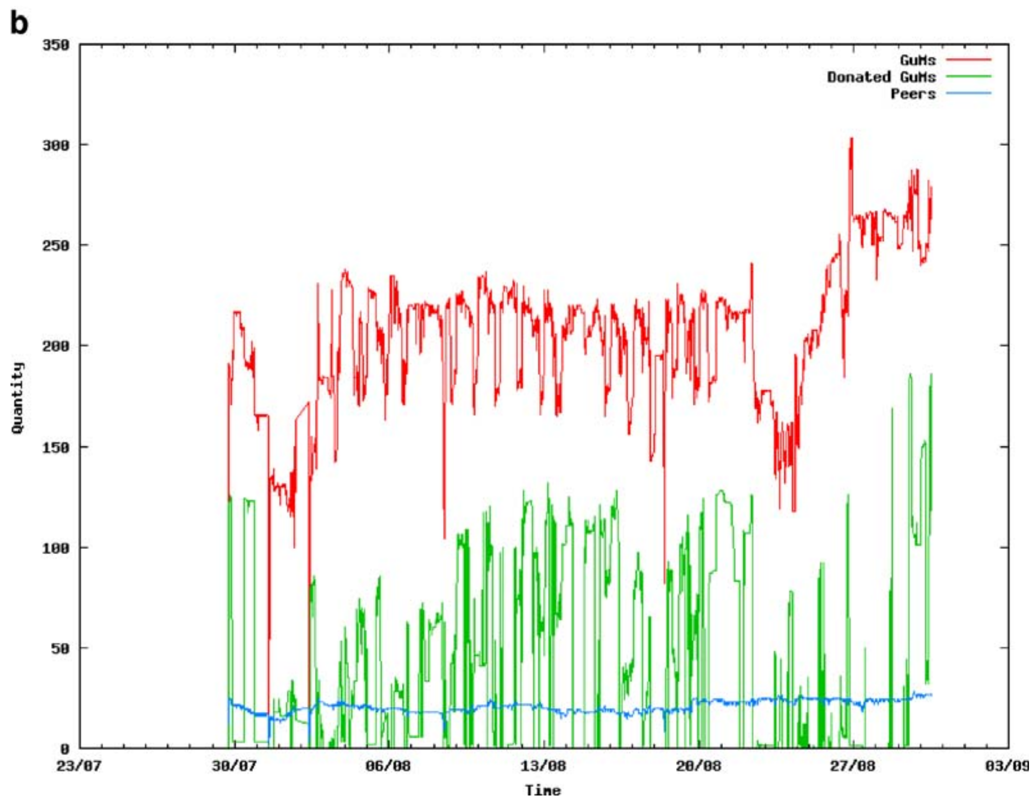
The use of OurGrid by numerous applications (molecular dynamics [62], simulations, climate forecast [21], imaging, hydrological management [56], and data mining [58]) provided us with very interesting feedback. A key point that became clear from the beginning is that dealing with failure in the Grid is very hard [46]. The key problem seems to be to identify the source fault that caused the failure. In fact, our users report that when a failure appears on the screen, it is very hard for the user to identify whether the problem is at his own application, somewhere in the Grid middleware, or even lower in the fabric that comprises the Grid. We argue that to overcome this problem, current Grid platforms must be augmented with multi-layer, collaborative diagnosis mechanisms [27]. We implemented such mechanisms in OurGrid by using automated tests to identify the root cause of a failure and propose the appropriate fix [27].

An interesting aspect of the deployment is the fact that we sometimes created solutions for problems that did not present themselves in practice. For example, in early versions, MyGrid implemented the proportional-share ticket-based

a



b



scheduler described in [18]. This scheduler is technically very interesting. It allows a user who runs more than one application simultaneously to define what fraction of resources obtained from the Grid should be allocated to each application. However, none of our users found this capability useful.

Another aspect of user behavior that did not match our expectations was the way in which users interact with the system. We designed OurGrid so that a user can describe a job in a very simple scripting language, relying mainly on the file abstraction (as described in Section 5). In fact, OurGrid can execute multiple independent runs of an application even when the user does not have the application's source code, only the binaries. All that is needed is that such an application reads the input from files, and produces the output to files. We thought that this simplicity would greatly ease the use of OurGrid. Nevertheless, the vast majority of users decided to use OurGrid's Java API for job submission.

Finally, rather to our dismay, a few labs that installed OurGrid asked for control on the set of other labs with whom they would interact. Some labs would like to configure the set of labs with whom they exchange favors. Other labs would like to give preferential treatment to their pals. While this goes against the ideal of an open free-to-join community, the Network of Favors (see Section 3) treats this gracefully. If there is a lab A that specifies the labs to whom it will donate resources, and there is a lab B that is not one of A's favorites, B's favors to A will not be reciprocated and B will soon stop interacting with A.

As well as bringing surprises, real use also brings new requirements to be added to the software. The most common requirement is checkpointing. Since local jobs kill foreign jobs, a user with very long running tasks has a very small chance of finishing a task on a foreign machine. We are using the Condor checkpoint library to cope with this. Another common requirement is the ability to restart OurGrid's components without losing any work. We are investigating whether saving the components' state in an embedded database addresses well this issue. Finally, some users have requested the ability for tasks to communicate, enabling applications more sophisticated than BoT. Once we have moved from RMI to Jabber, we intend to export Jabber resource endpoints to OurGrid applications.

8. Related Work

Grid computing is a very active area of research [11, 34]. Although it started within High Performance Computing, people have realized that Grid technology could be used to deliver computational services on demand. This observation has brought about convergence between Grid and Web Services technologies, as seen in standards like OGSA/OGSI [61] and its successor WSMF [23]. These standards are currently being implemented by both academia and industry. Most notably, these standards are being implemented by Globus [37], maybe the project with greatest visibility in Grid Computing.

However, this mainstream work in Grids has evolved more towards control assurance and interoperability with commercial standards. Our work is more concerned with scalability and simplicity. Condor and voluntary computing systems (such as SETI@home, BOINC, XtremWeb and Bayanihan) are closer to our work.

Condor was initially conceived for campus-wide networks [43], but has been extended to serve as a broker in a Globus Grid [32], as well as to allow the federation of multiple Condors in a Flock of Condors [15, 28]. In fact, a Flock of Condors and OurGrid are alike in the sense that they federate multiple distinct administrative domains that want to share their resources. The key difference between OurGrid and Condor is that "Condor assumes that a fair amount of trust exists between machine owners and users that wish to run jobs" [60], whereas OurGrid assumes the opposite. This difference in assumption affects most components of both solutions. OurGrid, for example, uses much stronger sandboxing technology than Condor. This comes at the expense that OurGrid applications are (currently) limited to Bag-of-Tasks, whereas Condor applications are not. Also, OurGrid places great effort in encouraging the labs (i.e. different administrative domains) to share their idle resources with labs they do not even know (since the more resources a lab

donates, the more resources it is likely to receive when needed). The Flock of Condors, on the other hand, does not address the motivation for different labs to share their resources. It was designed to allow for small-scale flocking among labs that already know and trust each other. OurGrid was designed to make it technically and administratively possible to scale up much larger than that.

Condor and OurGrid create Grids in which resource providers and resource consumers are roles played by the same people. As an alternative, voluntary computing efforts suggest a more asymmetrical view, in which many people voluntarily donate resources to a few projects of great public appeal. Arguably, voluntary computing originated from the huge success achieved by SETI@home [2, 57]. SETI@home makes no distinction between the application itself (search of extraterrestrial intelligence evidence in radio signals) and its Grid support. On the other hand, BOINC [3] has been introduced as a sequel to SETI@home, promising exactly such a separation. BOINC aims to create a voluntary computing infrastructure that can be used by several different applications. XtremWeb [29] also provides a similar platform. It places special care in the provision of a fault-tolerant programming environment for general parallel applications [12]. The Bayanihan project also aims to create a voluntary computing infrastructure and includes a very interesting contribution to tolerating sabotage (i.e. bogus volunteer results) [54]. In fact, we plan to use the Bayanihan approach to deal with sabotage, as mentioned in Section 4.

In principle, however, nothing precludes using OurGrid for voluntary computing. Labs could promote the voluntary installation of OurGrid worker machines (e.g. our native implementation of GuM) connected to their peers. In this scenario, voluntary machines act as local machines of a given lab (to which they donate their idle cycles). Since local jobs preempt foreign jobs, local jobs always have priorities on the machines donated to a given lab. However, if the lab temporarily does not have enough demand for its resources (owned and volunteered), the surplus is donated to the OurGrid community at large. This ensures that when, in the future, the lab computational demand is greater than its resources, it will have a greater chance of getting resources from other peers.

It is interesting to notice that OurGrid's underlying principle that a peer prioritizes other peers from whom it has received resources also provides the rationale for BitTorrent's incentive scheme for sharing bandwidth in peer-to-peer systems [20]. However, OurGrid's incentive scheme is designed for sharing computation rather than bandwidth, and is considerably different from BitTorrent's. First, Bittorrent's algorithm only considers very recent behavior. Second, in BitTorrent you do not diminish the local reputatation of a peer when you do it a favor. Third, the optimistic-unchoking part of the BitTorrent algorithm favors peers who have donated nothing over peers who have donated a small amount.

9. Conclusions

OurGrid is an open free-to-join collaborative Grid that caters for Bag-of-Tasks applications. OurGrid is in production since December 2004 and its current status can be seen at status.ourgrid.org. Labs wanting to join OurGrid just download the software from http://www.ourgrid.org and install it. Joining is automatic; no paperwork or approvals of any sort are required. The vision is that OurGrid provides a massive worldwide computing platform on which research labs can trade their spare compute power for the benefit of all.

In order to realize the OurGrid vision, we made two major contributions to the state of art in Grid computing. First, as shown in Section 3, the Network of Favors and its associated Autonomous Accounting assures fairness for the labs participating in the Grid by dealing with freeriders in a totally decentralized and autonomous way. Being able to promote collaboration without relying on centralized infrastructure is key to making OurGrid feasible and easy to deploy. Second, as presented in Section 5, we devised schedulers that achieve good performance without using information about the Grid or the application. These schedulers use task replication to deal with unfortunate task-to-processor allocations. Although consuming more resources, these schedulers simplify matters for the user (who does not have to estimate application execution times) and greatlysimplify the design and deployment of

OurGrid (because we avoid the need for a scalable and accurate infrastructure for monitoring and forecasting performance).

Naturally, OurGrid keeps evolving to be simpler, faster, and more complete, as discussed throughout this paper. We highlight as critical aspects where improvement is needed (i) making sandboxing more convenient, and (ii) improving the system's ability to deal with large amounts of data. In the end, our goal is to regain the simplicity and power of the original Grid motto: "Plug into the Grid and solve your problem."

Acknowledgments Many thanks to the OurGrid team (http://www.ourgrid.org/twiki-public/bin/view/OG/OurPeople). This work would have not happen without everybody's commitment to do their best. In particular, thanks to Ayla Dantas for the fabulous coaching of the team, Érica Gallindo for the great integration work, and José Pergentino for the help in generating the usage graphs. Thanks also to the OurGrid users and contributors. You are the ones who make this project alive. Francisco Brasileiro and Walfredo Cirne would like to thank the partial financial support from CNPq/Brazil (grants 300.646/96 and 302.317/03, respectively). Thanks also to Katia Saikoski for the comments and suggestions.

References

1. Abramson, D., Buyya, R., Giddy, J.: A computational economy for Grid computing and its implementation in the Nimrod-G resource broker. Future Gener. Comput. Syst. **18**, 1061–1074 (2002)
2. Anderson, D., Cobb, J., Korpela, E.: SETI@home: An experiment in public-resource computing. Communications of the ACM **45**(11), 56–61 (2002)
3. Anderson, D.: Public computing: Reconnecting people to science. Proceedings of Shared Knowledge and the Web, Madrid, Spain, Nov. 17–19 2003
4. Andrade, N., Brasileiro, F., Cirne, W., Mowbray, M.: Discouraging free-riding in a peer-to-peer CPU-sharing Grid. Proceedings of 13th IEEE International Symposium on High-Performance Distributed Computing (HPDC13), Honolulu, Hawaii, 4–9 June 2004
5. Andrade, N., Cirne, W., Brasileiro, F., Roisenberg, P.: OurGrid: An approach to easily assemble Grids with equitable resource sharing. Proceedings of 9th Workshop on Job Scheduling Strategies for Parallel Processing, June 2003
6. Andrade, N., Mowbray, M., Cirne, W., Brasileiro, F.: When can an autonomous reputation scheme discourage free-riding in a peer-to-peer system? Proceedings of 4th Workshop on Global and Peer-to-Peer Computing (GP2PC), Chicago, USA, 19–22 April 2004
7. AspectJ Team: The AspectJ Programming Guide. http://www.eclipse.org/aspectj/doc/released/progguide/index.html. Cited 14 March 2006 (2006)
8. Barham, P., et al.: Xen and the art of virtualization. Proceedings of SOPS'2003
9. Basu, S., Banerjee, S., Sharma, P., Lee, S.-J.: NodeWiz: Peer-to-peer resource discovery for Grids. Proceedings of 5th International Workshop on Global and Peer-to-Peer Computing (in conjunction with CCGRID 2005), May 2005
10. Beck, K.: Extreme Programming Explained: Embrace Change. Addison-Wesley (1999)
11. Berman, F., Fox, G., Hey, T. (eds.): Grid Computing: Making The Global Infrastructure a Reality. Wiley (2003)
12. Bosilca, G., et al.: MPICH-V: Toward a scalable fault tolerant MPI for volatile nodes. In: Proceedings of 2002 ACM/IEEE Conference on Supercomputing, Baltimore, Maryland, pp. 1–18, 2002
13. Buyya, R., Abramson, D., Giddy, J.: An economy driven resource management architecture for computational power Grids. Proceedings of International Conference on Parallel and Distributed Processing Techniques and Applications, 2000
14. Buyya, R., Vazhkudai, S.: Compute Power Market: Towards a market-oriented Grid. Proceedings of 1st IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2001), Beijing, China, 2001
15. Butt, A.R., Zhang, R., Hu, Y.: A self-organizing flock of condors. Proceedings of Supercomputing, 2003
16. Casanova, H., et al.: Heuristics for scheduling parameter sweep applications in Grid environments. In: Proceedings of 9th Heterogeneous Computing Workshop, pp. 349–363, 2000
17. CERN: Worldwide LCG Computing Grid. http://lcg.web.cern.ch/LCG/. Cited 14 March 2006 (2006)
18. Cirne, W., Marzullo, K.: The Computational Co-op: Gathering Clusters into a Metacomputer. Proceedings of IPPS/SPDP'99, April 1999
19. Cirne, W., et al.: Running bag-of-tasks applications on computational Grids: The MyGrid approach. Proceedings of ICCP'2003: International Conference on Parallel Processing, Oct. 2003
20. Cohen, B.: Incentives build robustness in BitTorrent. Proceedings of Workshop on Economics of Peer-to-Peer Systems, June 2003
21. COPAD project announcement. http://www.eradigital.com.br/clientes/ourgrid/news01.shtml. Cited 14 March 2006
22. Costa, L., Cirne, W., Fireman, D.: Converting space shared resources into intermittent resources for use in bag-of-tasks Grids. Proceedings of 17th Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'2005), Oct. 2005
23. Czajkowski, K., et al. From Open Grid Services Infrastructure to WS-Resource Framework: Refactoring & Evolution. Version 1.1, 3/05/2004. http://www.globus.org/wsrf/specs/ogsi_to_wsrf_1.0.pdf. Cited 14 March 2006

24. Damiani, E., Vimercati, S., Paraboschi, S., Samarati, P.: Managing and sharing servents' reputations in peer-to-peer systems. IEEE Transactions on Data and Knowledge Engineering **15**(4), 840–854 (2003)
25. To appear: Dantas, A., Cirne, W., Saikoski, K.: Using AOP to Bring a Project Back in Shape: The OurGrid Case. J Braz Comput Soc. Available at http://walfredo.dsc.ufcg.edu.br/resume.html#publications. Cited 14 March 2006
26. Dodonov, E., Sousa, J., Guardia, H.: GridBox: Securing hosts from malicious and greedy applications. In: Proceedings of Middleware for Grid Computing, pp. 17–22, 2004
27. Duarte, A., Brasileiro, F., Cirne, W., Alencar-Filho, J.: Collaborative fault diagnosis in Grids through automated tests. Proceedings of 20th International Conference on Advanced Information Networking and Applications (AINA'2006), April 2006
28. Epema, D., et al.: A worldwide flock of condors: Load sharing among workstation clusters. Future Gener. Comput. Syst. **12** (1996)
29. Fedak, G., et al.: XtremWeb: A generic global computing system. In: Proceedings of 1st International Symposium on Cluster Computing and the Grid, Brisbane, Australia, pp. 582–587, 2001
30. Feitelson, D.: Parallel Workloads Archive. http://www.cs.huji.ac.il/labs/parallel/workload/. Cited 14 March 2006
31. Figueiredo, R., Dinda, P., Fortes, J.: A Case for Grid Computing on Virtual Machines. Proceedings of Intl. Conf. on Distributed Computing Systems (ICDCS), 2003
32. Frey, J., et al.: Condor-G: A computation management agent for multi-institutional Grids. Proceedings of 10th IEEE Symposium on High Performance Distributed Computing, HPDC'10, San Francisco, California, August 7–9, 2001
33. Foster, I., Iamnitchi, A.: On death, taxes, and the convergence of peer-to-peer and Grid computing. Proceedings of 2nd International Workshop on Peer-to-Peer Systems (IPTPS'03), Berkeley, California, Feb 2003
34. Foster, I., Kesselman, C. (eds.): The Grid: Blueprint for a New Computing Infrastructure, 2nd edn. Morgan Kaufmann, 2004
35. Garfinkel, T.: Traps and pitfalls: Practical problems in system call interposition based security tools. Proceedings of Internet Society's 2003 Symposium on Network and Distributed System Security (NDSS 2003)
36. Garfinkel, T., et al.: Terra: A virtual machine-based platform for trusted computing. Proceedings of 19th Symposium on Operating System Principles (SOSP 2003)
37. The Globus alliance: http://www.globus.org. Cited 14 March 2006
38. Hughes, D., Coulson, G., Walkerdine, J.: Free riding on gnutella revisited: The bell tolls? IEEE Distrib. Syst. Online **6**(6) (2005)
39. Jabber, Inc.: http://www.jabber.org/. Cited 14 March 2006
40. Kamvar, S., Schlosser, M., Garcia-Molina, H.: EigenRep: Reputation management in peer-to-peer networks. Proceedings of 12th International World Wide Web Conference, Budapest, Hungary, May 2003
41. Kondo, D., Chien, A., Casanova, H: Resource management for short-lived applications on enterprise desktop Grids. Proceedings of Supercomputing'2004, Pittsburgh, Pennsylvania, Nov. 2004
42. Lee, C., et al.: Are user runtime estimates inherently inaccurate? Proceedings of 10th Job Scheduling Strategies for Parallel Processing, June 2004
43. Litzkow, M., Livny, M., Mutka, M.: Condor: A hunter of idle workstations. Proceedings of 8th International Conference of Distributed Computing Systems, pp. 104–111, June 1988
44. Loscocco, P., et al.: The inevitability of failure: The flawed assumption of security in modern computing environments. In: Proceedings of 21st National Information Systems Security Conference, pp. 303–314, Oct. 1998
45. Loscocco, P., Smalley, S.: Integrating flexible support for security policies into the linux operating system. Proceedings of FREENIX track of USENIX Annual Technical Conference, June 2001
46. Medeiros, R., Cirne, W., Brasileiro, F., Sauvé, J.: Faults in Grids: Why are they so bad and what can be done about it? Proceedings of Grid 2003: 4th International Workshop on Grid Computing, November 2003
47. Grid Economic Services Architecture Working Group. http://www.doc.ic.ac.uk/~sjn5/GGF/gesa-wg.html. Cited 14 March 2006
48. Paranhos, D., Cirne, W., Brasileiro, F.: Trading cycles for information: Using replication to schedule bag-of-tasks applications on computational Grids. Proceedings of Europar'2003, Austria, 2003
49. Riehle, D., Fraleigh, S., Bucka-Lassen, D., Omorogbe, N.: The architecture of a UML virtual machine. In: Proceedings of Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '01), pp. 327–341, 2001
50. Ripeanu, M., Foster, I.: Mapping the gnutella network: Macroscopic properties of large-scale peer-to-peer systems. Proceedings of First International Workshop on Peer-to-Peer Systems (IPTPS), 2002
51. Ripeanu, M.: The Globus Toolkit Ecosystem (and How to Make it Work for You). http://people.cs.uchicago.edu/~matei/GlobusEcosystem/. Cited 14 March 2006
52. Santos, R., Andrade, A., Cirne, W., Brasileiro, F., Andrade, N.: Accurate autonomous accounting in peer-to-peer Grids. In: Proceedings 3rd Workshop on Middleware for Grid Computing (MGC2005), November 2005
53. Santos-Neto, E., Cirne, W., Brasileiro, F., Lima, A.: Exploiting replication and data reuse to efficiently schedule data-intensive applications on Grids. In: Proceedings of 10th Workshop on Job Scheduling Strategies for Parallel Processing, June 2004
54. Sarmenta, L: Sabotage-tolerance mechanisms for volunteer computing systems. Future Gener. Comput. Syst. **18**(4) (2002)

55. Saroiu, S., Gummadi, P., Gribble, S.: A measurement study of peer-to-peer file sharing systems. Proceedings of Multimedia Computing and Networking (MMCN) 2002, San Jose, California, Jan. 2002
56. SegHidro project team: SegHidro Web Site http://seghidro.lsd.ufcg.edu.br/. Cited 14 March 2006
57. SETI@home team: SETI@home statistics web page. http://setiathome.ssl.berkeley.edu/totals.html. Cited March 2005
58. Silva, F., et al.: Running data mining applications on the Grid: A bag-of-tasks approach. Proceedings of International Conference on Computational Science and its Applications, 2004
59. Son, S., Livny, M.: Recovering internet symmetry in distributed computing. Proceedings of GAN'03 Workshop on Grids and Advanced Networks, Tokyo, Japan, 12–15 May 2003
60. Thain, D., Tannenbaum, T., Livny, M.: Distributed Computing in Practice: The Condor Experience. Concurrency and Computation: Practice and Experience **17**(2–4), 23–356 (2005)
61. Tuecke, S., et al.: Open Grid Services Infrastructure (OGSI) Version 1.0. Global Grid Forum Draft Recommendation, 6/27/2003. http://www.globus.org/toolkit/draft-ggf-ogsi-gridservice-33_2003-06-27.pdf. Cited 14 March 2006
62. Veronez, C., Osthoff, C., Pascutti, P.: HIV-I Protease mutants molecular dynamics research on Grid computing environment. In: Proceedings of WOB pp. 161–164, 2003