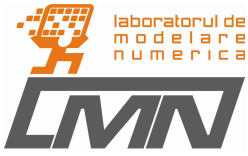

Universitatea "Politehnica" din Bucureşti

Facultatea de Inginerie Electrica –CIEAC-LMN

Marie Curie Doctoral School in EE and CS

Spl. Independenţei 313, RO-060042, Bucureşti ROMANIA

http://www.lmn.pub.ro/ Tel/Fax: (40 21) 311 8004, (40 21) 316 95 71,

e-mail: lmn@lmn.pub.ro

Report title: Scheduling algorithms for desktop grid platforms

Author: Bogdan Bogdanov

Scientific supervisor: prof. Daniel Ioan

Date: 06.08 2009

Acknowlidgement

I would like to express my sincere gratitude to my advisor Prof. Daniel IOAN for providing me his invaluable insights, encouragement, guidance and financial support throughout my whole study period of this internship.

Contents

I Computational Grids

The popularity of the Internet and the availability of powerful computers and high-speed networks as low-cost commodity components are changing the way we use computers today. This technology opportunity has led to the possibility of using networks of computers as a single, unified computing resource. It is possible to cluster or couple a wide variety of resources including supercomputers, storage systems, data sources, and special classes of devices distributed geographically and use them as a single unified resource, thus forming what is popularly known as a Computational Grid.

In particular, Grid computing can be defined as the coordinated resource sharing and problem solving in dynamic, multi-institutional collaborations. More simply, Grid computing typically involves using many resources (computer, data, I/O, instruments, etc.) to solve a single, large problem that could not be executed on any one resource. As a matter of fact, various Grid application scenarios have been explored within both science and industry. These applications include compute-intensive, data-intensive, sensor-intensive, knowledge-intensive and collaboration-intensive scenarios and address problems ranging from fault diagnosis in jet engines and earthquake engineering to bioinformatics, biomedical imaging, and astrophysics.

The applications cited above need a coordinated resource sharing, where the sharing is not primarily file exchange but rather direct access to computers, software, data, and other resources, as is required by a range of collaborative problem-solving and resource-brokering strategies emerging in industry, science and engineering. Thus, this sharing is, necessarily, highly controlled, with resource provides and consumers defining clearly and carefully just what is shared, who is allowed to share, and the conditions under which sharing occurs. A set of individuals and/or institutions defined by such sharing rules form what is called a virtual organization (VO).

There are three main issues that characterize Computational Grids: heterogeneity, scalability and dynamic adaptability. A Grid involves a multiplicity of resources that are heterogeneous in nature and might span numerous administrative domains across wide geographical distances. Moreover, a Grid might grow from few resources to millions. This raises the problem of potential performance degradation as the Grid size increases. Consequently, applications that require a large number of geographically located resources must be designed to be extremely latency tolerant. Finally, in a Grid, resource failures are the rule, not the exception. In fact,

with so many resources in a Grid, the probability of some resource failing is naturally high.

Moreover, the participating hosts can be reclaimed by the respective owners at any time, and is impossible to know in advance whether and when they will become available again. The resource managers or applications must tailor their behaviour dynamically so as to extract the maximum performance from the available resources and services.

In spite of these problems, Grid computing must provide an easy access to a virtually unlimited computing and distributed data resources, so it must be able to discover, allocate, negotiate, monitor, and manage the use of network-accessible capabilities in order to achieve various end-to-end or global qualities of service. All those activities can be considered as parts of a global task called resource management.

In traditional computing systems, resource management is a well-studied problem. Resource managers such as batch schedulers, workow engines, and operating systems exist for many computing environments. These resource management systems are designed and operate under the assumption that they have completed control of a resource and thus can implement the mechanisms and policies needed for efective use of that resource in isolation.

Unfortunately, this assumption does not apply to Grids because of
the main issues previously described. This situation is complicated by the general lack of data available about the current system and the competing needs of users, resource owners and administrators of the system.

For this reason, much of the early work in Grid resource management focused on overcoming these basic issues of heterogeneity, for example through the definition of standard resource management protocols and standard mechanisms for expressing resource and task requirements.

The initial challenges of Grid computing concerning how to run a job, how to transfer large files, how to manage multiple user accounts on different systems have been resolved to first order, so users and researchers can now address the issues that will allow more e_ective usage of the resources.

Significant challenges remain, however, in understanding how these mechanisms can be e_ectively combined to create seamless virtualized views of underlying resources and services. Some of these challenges lie strictly within the domain of resource management, for example, robust distributed algorithms for negotiating simultaneous service level agreements across a set

of resources. Other issues, such as expression of resource policy for purposes of discovery and enhanced security models that support exible delegation of resource management to intermediate brokers are closely tied to advances in other aspects of Grid infrastructure. Hence, the key to progress in the coming years is to create an extensible and open infrastructure that can incorporate these advances as they become available.

Computational Grids are becoming the main execution platform for high performance and distributed applications. The successful of Grid paradigm is also due to the growth of the World Wide Web (WWW) and exploding popularity of the Internet that has created a new much large scale opportunity for distributed computing. As a matter of fact, millions of desktop PCs are connected to wide-area networks both in the enterprise and in the home. The exploitation of idle cycles on pervasive desktop PC systems has recently received much attention from the scientific community. This new platform for high throughput applications is called Desktop Grids . The details of this platform and the relative problems will be described in the next section.

II Desktop Grids

The world's computing power and disk space is no longer primarily concentrated in supercomputer centers and machine rooms, but is distributed in hundreds of millions of personal computers and game consoles belonging to the general public. Desktop Grids uses these resources to do scientific supercomputing. The number of Internet-connected PCs is indeed growing rapidly, and is projected to reach 1 billion by 2015. Together, these PCs could provide many PetaFLOPs of computing power. The public resource approach applies to storage as well as computing. If 100 million computer users each provide 10 Gigabytes of storage, the total (one Exabyte, or 1018 bytes) would exceed the capacity of any centralized storage system. This paradigm enables previously infeasible research, encourages public awareness of current scientific research, catalyzes global communities centered around scientific interests, and gives the public a measure of control over the directions of scientific progress.

Public-resource computing emerged in the mid-1990s with two projects, namely the Great Internet Mersenne Prime Search(GIMPS) and Distributed net. In 1999 another project, called SETI@home, attracted millions of participants worldwide, providing a sustained

processing rate of over 70 TeraFLOPS (in contrast, the largest conventional supercomputer at that time, the NEC Earth Simulator, provided about 35 TeraFLOPs).

SETI@home is a scientific experiment that uses Internet-connected computers
in the Search for Extraterrestrial Intelligence (SETI), Desktop Grid and Grid computing share the goal of better utilizing existing computing resources. However, there are profound differences between the two paradigms. As a matter of fact, Grid computing involves organizationally-owned resources: supercomputers, clusters, and PCs owned by universities, research labs, and companies. These resources are centrally managed by IT professionals, are powered on most of the time, and are connected by full-time, high-bandwidth network links. Furthermore, there is a symmetric relationship between organizations: each one can either provide or use resources. In contrast, Desktop Grids involve an asymmetric relationship between projects and participants. Projects are typically small academic research groups with limited computer expertise and manpower. Most participants are individuals who own Windows, Macintosh and Linux PCs, connected to the Internet by telephone or cable modems or DSL, and often behind network-address translators (NATs) or firewalls. The computers are frequently turned off or disconnected from the Internet. Participants are not computer experts, and participate in a project only if they are interested in it and receive incentives such as credit and screensaver graphics. Projects have no control over participants, and cannot prevent malicious behavior.

A second difference is that, Grid computing has many requirements that Desktop Grid computing does not. A Grid architecture must accommodate many existing commercial and research-oriented academic systems, and must provide a general mechanism for resource discovery and access. In fact, it must address all the issues of dynamic heterogeneous distributed systems, an active area of Computer Science research for several decades. This has led to architecture such as Open Grid Services Architecture (OGSA) , which achieves generality at the price of complexity and, to some extent, performance.

In contrast, the main characteristic of Desktop Grid systems is the unobtrusiveness because the resource used are installed and designed for purposes other than distributed computing (e.g. desktop word processing, web information access, spreadsheet, etc.), thus the resource must be exploited without disturbing their primary use. Moreover, the machine including its data, hardware, and processes must be protected from a misbehaving Desktop Grid applications. Analogously, the application's executable, input, and

output data, which may be proprietary, must be protected from user inspection and corruption.

As cited in the previous section, another requirement is concerned with scalability: Desktop Grids must scale to the 1,000's, 10,000's, and 100,000's of desktop PC's deployed in enterprise networks. Systems must scale both upward and downward performing well with reasonable effort at a variety of system scales. With thousands to hundreds of thousands of computing resources, management and administration effort in a Desktop Grid cannot scale up with the number of resources. Desktop Grid systems must achieve manageability, that is the systems should provide tools for installing and updating users easily, and tools for managing applications and resources, and monitoring their progress.

Finally, the last two requirements essentially for any Desktop Grid systems are efficiency (the system must exploit all the idle resource available) and robustness: the system must be tolerant of both server failure (for example, data server crashes) and user failure (for example, the user shutting off his/her machine). Conventionally, the term failure refers to a defect of hardware or software. We use the term failure broadly to include all causes of task failure, including not only failure of the host's hardware or software, but also, for example, keyboard/mouse activity that causes the user to kill a running task.

III Desktop Grid platforms

In this section, we present a overview of four representative Desktop Grid projects: BOINC, Entropia, XtremWeb and OurGrid.

1. Berkeley Open Infrastructure for Network Computing

BOINC (Berkeley Open Infrastructure for Network Computing) is an open source platform for Desktop Grid computing. BOINC is being developed at U.C. Berkeley Spaces Sciences Laboratory by the group that developed and continues to operate SETI@home. BOINC's general goal is to advance the Desktop Grid computing

paradigm by encouraging the creation of many projects and a large fraction of the world's computer owners to participate in one or more projects.

A BOINC project corresponds to an organization or research group that does Desktop Grid computing. It is identi_ed by a single master URL, which is the home page of its web site and also serves as a directory of scheduling servers. Participants register with projects. A project can involve one or more applications, and the set of applications can change over time. The server complex of a BOINC project is centered around a relational database that stores descriptions of applications, platforms, versions, tasks, results, accounts, teams, and so on. Server functions are performed by a set of web services and daemon processes: scheduling servers handles RPCs from clients; they issue work and handle reports of completed results.

Data servers handle file uploads using a certificate-based mechanism to ensure that only legitimate _les, with prescribed size limits, are uploaded. File downloads are handled by plain HTTP. BOINC provides tools (Python scripts and C++ interfaces) for creating, starting, stopping and querying projects adding new applications, platforms, and application versions, creating tasks, and monitoring server performance. Moreover, BOINC is designed to be used by scientists, not by system programmers or IT professionals; the tools are simple and well-documented, and a full-featured project can be created in a few hours.

To prevent erroneous computational results, BOINC uses a technique called redundant computing. In particular, a project can specify that N results should be created for each task. Once M _ N of these have been distributed and completed, an application-specific function is called to compare the results and possibly select a canonical result. If no consensus is found, or if results fail, BOINC creates new results for the task, and continues this process until either a maximum result count or a timeout limit is reached.

As cited in the previous section, scalability is another problem that must be consider in Desktop Grid computing projects. Specially, in contest where we may have millions of participants and a relatively modest server complex. As a matter of fact, if all the participants simultaneously try to connect to the server, a disastrous overload condition will generally develop. BOINC has a number of mechanisms to prevent this. In particular, all client/server communication uses exponential backoff in the case of failure. Thus, if a BOINC server comes up after an extended outage, its connection rate will be the longterm average. The exponential backoff scheme is extended to computational errors as well. If, for some reason, an application fails immediately on a given host, the BOINC client will not repeatedly contact the server; instead, it will

delay based on the number of failures. BOINC is being used by several existing projects (SETI@home, Predictor@ home, climateprediction.net) and by several other projects in development. Many areas of the BOINC design are incomplete. For example, some projects require e_cient data replication: Einstein@home uses large (40 MB) input _les, and a given input _le may be sent to a large number of hosts (in contrast with projects like SETI@home, where each input file is different). In its initial form, Einstein@home sends the _les separately to each host, using a system of replicated data servers. In order to improve the effciency of the data replication, the BOINC's team is planning to exploit a mechanism such as BitTorrent based on peer-to-peer communication.

2. Entropia

Key advantages of the Entropia system are the ease of application integration and a new model for providing security and unobtrusiveness for the application and client machine. To provide rapid application integration, Entropia uses a binary modification technology that obviates access to the applications source code while providing strong security guarantees and ensuring unobtrusive application execution. Other systems require developers to modify their source code to use custom APIs or simply rely on the application to be well behaved and provide weaker security and protection.

However, it is not always possible to get access to the application source code (especially for commercial applications) and maintaining multiple versions of the source code can require a significant continuous development effort. As to relying on the good intentions of the application programmers, the experiments show that even commonly used applications in use for quite some time can at times exhibit anomalous behavior. Entropia's approach ensures both a large base of potential applications and a high level of control over the application's execution.

The Entropia system addresses the requirements described in Section by aggregating the raw desktop resources into a single logical resource. This logical resource is reliable, secure and predictable despite the fact that the underlying raw resources are unreliable (machines may be turned off or rebooted), insecure (untrusted users may have electronic and physical access to machines) and unpredictable (machines may be heavily used by the desktop user at any time). This logical resource provides high performance for applications through parallelism while always respecting the desktop user and his or her use of the desktop machine. Furthermore, this logical resource can be managed from a

single administrative console. Addition or removal of desktop machines from the Entropia system is easily achieved, providing a simple mechanism to scale the system as the organization grows or as the need for computational cycles grows.

To support the execution of a large number of applications, and to support the execution in a secure manner, Entropia employs a proprietary binary sandboxing techniques that enables any Win32 application to be deployed in the Entropia system with no modifications and no special system support. End-users of the Entropia system can use their existing Win32 applications and deploy them on the Entropia system in a matter of minutes.

This is significantly different than other early large-scale distributed computing systems like SETI@home and other competing systems that require rewriting and recompiling of the application source code to ensure safety and robustness.

Entropia's system architecture consists of three layers: a physical node management layer (that provides basic communication and naming, security, resource management, and application control), resource scheduling (that provides resource matching, scheduling and fault tolerant), and job scheduling. This architecture provides a modularity that allows each layer to focus on a smaller number of concerns, enhancing overall system capability and usability. This system architecture provides a solid foundation to meet the technical challenges as the use of distributed computing matures supporting the broadening the problems supportable by increasing the breadth of computational structure, resource usage, and ease of application integration. The implementation includes innovative solutions in many areas, but particularly in the areas of security, unobtrusiveness, and application integration. The system is applicable to a large number of applications such as virtual screening, sequence analysis, molecular modeling, and risk analysis. Unfortunately, it seems that there is no future for Entropia: this system is not longer maintained.

3. XtremWeb

XtremWeb is designed to provide a Global Computing framework for resolving different applications, on projects lead by institutions, commercial firms or open source communities.

The XtremWeb system may be seen as several sets of workers (ranging from workstations to PCs or even handheld devices) and servers (ranging from a stand alone PC to cluster of servers or distributed and specialized servers).

The XtremWeb system provides solutions to the desktop distributed computing challenges discussed in Section 1.2. In particular, XtremWeb targets high performance. Thus, although the workers protection suggests execution in a virtual environment, typically sandboxed Java bytecode, performance dictates that the end-user code should remain native. Like many other Global Computing systems, XtremWeb uses native code execution.

However, in contrary to them, XtremWeb allows any worker to execute different and downloadable applications. Currently, to become downloadable, the applications follow an ad-hoc verification process .Another important aspect is the unobtrusiveness. The user decides when XtremWeb may run a computation and what resources the computation may use. The availability of a given machine depends on the user presence (detected through the keyboard or mouse activity), the presence of noninteractive tasks (detected through the CPU, memory and I/O usage) and other conditions like night and day for instance. Resource utilization is continuously monitored by the worker. An interface to the resources is provided by Operating System features, e.g. the /proc directory for the Unix OSes. A user de_nes an availability policy simply by indicating for each resource a threshold above which the computer is usable for a Global Computation and a threshold that provokes the interruption of the computation. Controlling the resources used by the Global Computation can be tuned.

Besides the unobtrusiveness, the usability is another aspect that must be considered in a Desktop Grid system. As a matter of fact, in XtremWeb, the worker software comes with different tools to enforce its usability. The installation process follows two ways: installation on a single workstation and installation on a local area network. In both case it does not require the user to have special root privileges.

Finally, in order to provide the scalability, XtremWeb aims at scaling up to several hundreds of thousands of workers with corresponding performance improvement. This scalability goal is ensured by several design choices. First, the server throughput is increased by using cluster of servers and a meta server. Second, load balancing within the cluster of servers can be achieved through an external tool, able to sustain high throughput, typically via mechanisms supported by OS kernel (such as the Linux Virtual Server) or DNS aliases. Third, specialized servers can be dedicated to collecting the results so as to reduce the traffic to the root servers. Summary, the XtremWeb architecture exhibits several properties. By supporting native execution, XtremWeb can be used by institutions for setting up their own Global Computing system based on their legacy applications. Servers architecture enables scalability and fault tolerance. Future work will develop extensive testing on the

XtremWeb architecture to evaluate quantitatively its characteristics and limits. Tools have been developed to enforce the usability of the platform. Installation and administration over a LAN can be done in an automated way. The Web interface and contest between teams of users is an attractive way to involve the public. The first version of the worker software has been released for public download in November 2000. The Future work for XtremWeb is to find the relevant performance parameters that define and characterize a parallel architecture built through the Global Computing model.

4. OurGrid

The OurGrid system is defined as an open, free-to-join, cooperative grid which labs donate their idle computational resources in exchange for accessing other labs' idle resources when needed. The OurGrid scientists have implemented a system fast, simple, scalable and secure for Bag-of Tasks (BoT) grid applications (details about BoT will be discussed in the next section). In the rest of this section, we described how the goals of OurGrid have been implemented.

First of all, the OurGrid respects the fundamental characteristic of an Desktop Grid system: it must be non-intrusive, that is a local user has
priority for local resources. As a matter of fact, the submission of a local
job kills any foreign jobs that are running locally. Another key requirement
for Desktop Grids is scalability. OurGrid is scalable both in the sense that it supports thousands of labs, and that joining the system is straightforward.

OurGrid is based on a peer-to-peer network, where each labs in the Grid correspond to a peer in the system. The main problem using a peer-to-peer system is that the performance could be compromised by freeriders: a peer that only consume resources, never contributing back to the community.

In order to incentive peers to cooperate, and consequently discourage the freeriders, OurGrid uses a Network of Favors: a favor is the allocation of a processor to a peer that request 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. The rationale is that each peer autonomously prioritize donations to the peer to whom the owe most favors, motivating cooperation.

Since a given lab will commonly run tasks from other unknown labs, the security is also a fundamental aspect especially in

these days with so many software vulnerabilities. For this reason, OurGrid uses Sandboxing Without A Name (SWAN) to protect local resources from foreign unknown code. SWAN is a solution based on the Xen virtual machine that isolates the foreign code into a sandbox, where it cannot access local data nor use the network.

Finally, OurGrid has achieved the goals regarding simplicity and speed using MyGrid: a personal broker that performs application scheduling and provides a set of abstractions that hide the Grid heterogeneity from the user.

A efficient scheduler needs 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. MyGrid's first scheduler is Workqueue with Replication(WQR): a scheduling algorithm that uses no information about tasks or machines. In order to recover from bad allocations of tasks to machines (which are inevitable, since WQR uses no information), this algorithm uses replication. WQR will be described with more details in chapter V .

However, WQR does not take data transfer into account. For this reason, MyGrid provides another scheduling algorithm called Storage Affinity : it does not use dynamic, hard-to-obtain information as other Grid scheduler proposed. The idea is exploit data reutilization to avoid unnecessary data transfers. In particular, given a task the scheduler is able to calculate the storage affinity with respect to each other site. A site has a good storage affinity considering a particular task, if it contains data necessary to execute the task, that is it is not necessary to transfer data in the remote host in order to run the task. Since Storage Affinity does not use dynamic information, it can make "bad" assignment between tasks and resources. In order to recover from bad assignment, also Storage Affinity uses task replication strategy similar to WQR. 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. OurGrid is in production since December 2004 and its current status can be seen at http://status.ourgrid.org.

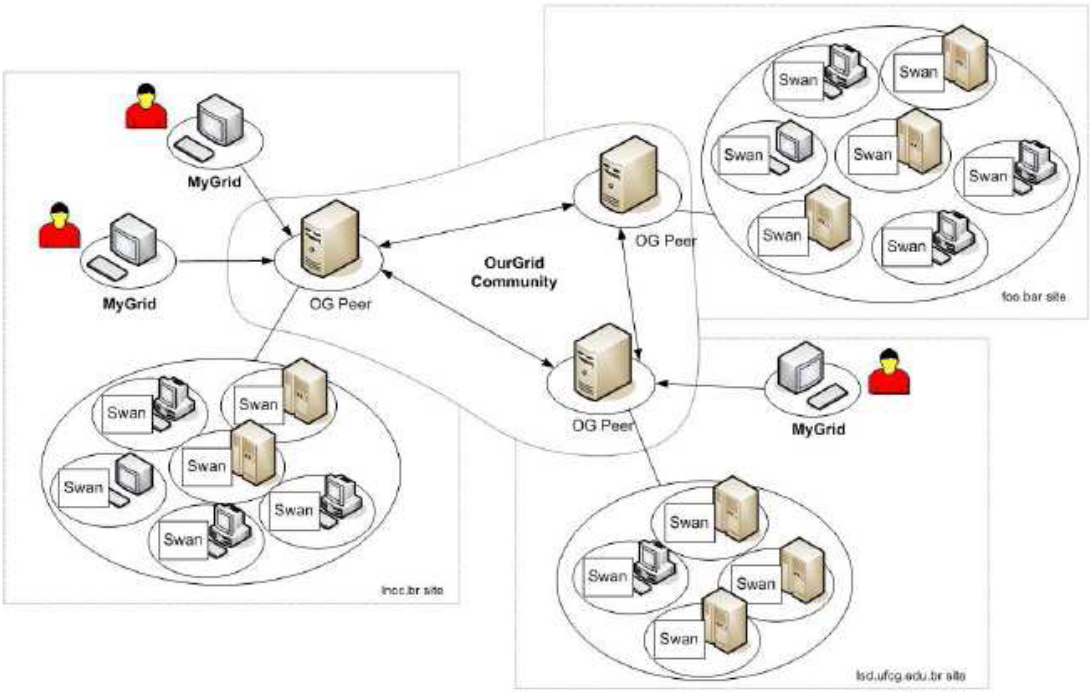

Figure 1.

IV Classification of scheduling algorithm categories

Scheduling algorithms							
Knowledge free				Knowledge based			
On-line		Batch		On-line		Batch	
Fault tolerant	No fault tolerant	Fault tolerant	No fault tolerant	Fault tolerant	No fault tolerant	Fault tolerant	No fault tolerant

1. Fault-tolerant scheduling

A Grid may potentially encompass thousands of resources, services, and applications that need to interact in order for each of them to carry out its task. The extreme heterogeneity of these elements gives rise to many failure possibilities, including not only independent failures of each resource, but also those resulting from interactions among them.

Moreover, resources may be disconnected from a Grid because of machine hardware and/or software failures or reboots, network misbehaviors, or process suspension/abortion in remote machines to prioritize local computations. Finally, configuration problems or middleware bugs may easily make an application fail even if the resources or services it uses remain available. In order to hide the occurrence of faults, or the sudden unavailability of resources, fault-tolerance mechanisms (e.g., replication or checkpointing-and-restart) are usually employed.

2. Knowledge-based and knowledge-free scheduling

The decisions a scheduler makes are only as good as the information provided to it. Many theoretical schedulers assume one has 100 percent of the information needed, at an extremely fine level of detail, and that the information is always correct. Unfortunately, as discuss later, this scenario is overly unrealistic. In general we have only the highest level of information. For example, it may be known that an application needs to run on Linux, will produce output _les somewhere between 20 MB and 30 MB, and should take less than three hours but might take as long as five. Or, it may be known that a machine is running Linux

and has a _le system located at certain address that ten minutes ago had 500 MB free, but there is no information about what will be free when one's application runs there.
Usually, the information regards the tasks (execution time, software and hardware requirements, etc) and the resources (computational power, availability, etc). In general, the scheduling algorithms assume a partial knowledge, and use it to calculate a sort of utility value for each assignment task/resource. With this value the scheduler is able to valuate how good is each assignment and decides how make the scheduling decisions. In some case, the goodness of an assignment could be decided also considering the cost to use a particular resource. As matter of fact, if the resources are freely usable any user could have an antisocial behavior (i.e., a single user can occupy the whole Computation Grid with his tasks). To avoid this situation, the scientific community has proposed some approaches based on micro economy theories.

3. On-line and batch-mode scheduling algorithms

Independently of the amount of information about resources and task knew, the scheduling policies can be grouped into two categories: on-line and batch mode. In the on-line mode, a task is assigned to a machine as soon as it arrives at the scheduler and this decision is not changed once it is computed.

Conversely, in the batch mode, tasks are not mapped onto the machines as they arrive; instead they are collected into a set that is examined for mapping at prescheduled times called mapping events. In the next section, we provide some example to put in evidenced the differences between on-line mode and batch mode scheduling.

V Examples

In this section, examples will be taken from the scientific literature to demonstrate their relationship to one another with respect to the taxonomy detailed in previous section.

1. Knowledge-free scheduling – WQR algorithm.

WQR is an extension of the classical WorkQueue (WQ) scheduling algorithm in which tasks in a bag are chosen in an arbitrary order and are sent to the processors as soon as they become available. WQR adds task replication to WQ in order to cope with task and host heterogeneity, as well as with dynamic variations of the available resource capacity due to the competing load caused by other Grid users. WQR works very similarly to WQ, in the sense that tasks are scheduled the same way. However, after the last task has been scheduled, WQR assigns replicas of already-running tasks to the processors that become free (in contrast, WQ leaves them idle). Tasks are replicated until a predefined replication threshold is reached. When a tasks replica terminates its execution,its other replicas are canceled.

By replicating a task on several resources,WQR increases the probability of running one of the instances on a faster machine, thereby reducing task completion time. WQR performance are equivalent to solutions that require full knowledge about the environment, at the expenses of consuming more CPU cycles.

Figure 3 shows in detail the WQR scheduling policy. This algorithm needs some data structures and functions. In particular, it uses two data structures: a queue of tasks to complete (represented by letter Q, line 2), a set of resources that compose our Computational Grid (represented by letter R, line 3) moreover, WQR uses a variable RPLFCT that represents the replication threshold (line 4) and four functions: RPL(t) (that returns the number of replicas created for task t, line 5), getRscFree(R) (that returns a identi_er of a resource available, line 6), deleteInstance(t) (that deletes all running instances of task t, line 7) and, _nally, assign(r,t) (that assigns task t to resource r, line 8). The algorithm starts checking the queue of tasks Q: if the queue of the tasks is not empty (line 10), the algorithm starts waiting (line 11) for two type of events: RscF ree or TaskDone. The _rst event (line 12) means that there is at least a resources idle, so it is ready to execute atask. Thus, the scheduler selects this resource r (line 13) and extracts a task t on the top of the queue Q (line 14). If the number of replicas created for task t is smaller than the replication threshold (line 15), a new instance of task t is created and submitted to resource r (line 16) and task t is inserted again in the queue Q (line 17). Conversely, if the event was TaskDone (line 19), all the running instances of task done are deleted (line 20). Thus, the hosts become ready to execute other tasks.

1: - data structures and functions -
2: Q {is the queue of the tasks (Qi is the i^{th} task)}

```
3: R {is the set of resources (Ri is the i^th resource)g
4: RPLFCT {is the maximum number of available replicas for each task}
5: RPL(t) {returns the number different instances provided}
6: getRscFree(R) {returns a resource available}
7: deleteInstances(t) {deletes all running instances of task t}
8: assign(r,t) {assigns task t to resource r}
9: - WQR algorithm -
10: while Q is not empty do
11:       wait(event)
12:         if (event = = "RscF ree") then
13:           r = getRscFree(R); {r is a resource available}
14:           t = pop front(Q); {extracts the first task t of the queue}
15:           if (RPL(t) < RPLFCT) then
16:             assign(r,t); {assigns task t to resource r}
17:             push back(Q,t); {adds task t to the end of the queue}
18:           end if
19:       else {event = = "TaskDone"}
20:         deleteInstances(t); {deletes all running instances of task t}
21:       end if
22: end while
```

Figure 3: WQR

2. On-line mode scheduling

In the on-line mode scheduling, the scheduler must select the best resource given a particular task. The resource selection is done ordering the hosts in the ready queue according to some criteria (e.g., by clock rate, by the number of cycles delivered in the past) and to assign tasks to the "best" hosts first. This method is also called resource prioritization.

The simplest on-line scheduling policy is First-Come-First-Serve (FCFS) where the scheduler selects the first host in the ready queue. Although this policy is not accurate since it does not consider any kind of information concerning either tasks or hosts, FCFS is often used also in important Desktop Grid projects like XtremWeb and BOINC. This is due to the fact that, in a dynamic and extremely heterogeneous environment like Grid it is difficult to obtain such information. Moreover many works, as described in, considers that when the number of tasks is about equal or greater than the number of hosts, there is just a little benefit of prioritization over FCFS. This can be explained considers that the most capable hosts tended to request tasks the most often, and so FCFS performed almost as well as any of the prioritization heuristic proposed in the scientific

literature. Many of these scheduling policies are based on the classical on-line scheduling algorithms: minimum completion time (MCT), minimum execution time (MET), switching algorithm (SA), k-percent best (KPB) and opportunistic load balancing (OLB).
For example, the work in [16] describes three methods for resource prioritization using different levels of information about the hosts from virtually no information to comprehensive historical statistics derived from trace for each host. In particular, for the PRI-CR method, hosts in the server's ready queue are prioritized by their clock rates. Similar to PRI-CR, PRICR- WAIT sorts hosts by clock rates, but the scheduler waits for a fixed period of 10 minutes before assigning tasks to hosts. The rationale is that collection a pool of ready hosts before making the assignments can improve host selection. Finally, the third method called PRI-HISTORY uses dynamic information, i.e. history of a host's past performance to predict its future performance. All the results report in [16] are based on scenarios where the number of tasks is comparable with the number of resources. We believe that di_erent scenarios where the number of tasks can be greater than the number of machines should be evaluated.
Other works considers the performance of prioritization methods limited
by the choice of the slowest hosts. For this reason they proposed other
methods based on the exclusion of some hosts and never use them to run application tasks (these methods are called resource exclusion). Filtering can be based on a simple criterion, such as hosts with clock rates below some threshold. Often, the distribution of resource clock rates is so skewed that the slowest hosts signi_cantly impede application completion, and so excluding them can potentially remove this bottleneck. A more sophisticated resource exclusion strategy consists in removing hosts that would not complete a task, if assigned to them, before some expected application completion time. In other words, it may be possible to obtain an estimate of when the application can reasonably complete, and not use any host that would push the application execution beyond this estimate. The advantage of this method compared to blindly excluding resources with a fixed threshold is that it should not be as sensitive to the distribution of clock rates. One of the scheduling algorithm belong at this category is real-time scheduling advisor (RTSA). RTSA is a system for scheduling soft real-time tasks using statistical predictors of host load. The system presents to a user the confidenceintervals for the running time of a task. These confidence intervals are formed using time series analysis of historical information about host load. However, the work assumes a

homogeneous environment and disregardtask failure caused by user activity, thus its relevance on Desktop Grids is questionable.
The work described is the most relevant in terms of Desktop Grid scheduling. The author investigates the problem of scheduling multiple independent compute bound applications that have soft-deadline constraints on the Condor Desktop Grid system. Each "application" in this study consists of a single task. The issue addressed in the paper is how to prioritize multiple applications having soft deadlines so that the highest number of deadlines can be met. The author uses two approaches. One approach is to schedule the application with the closest deadline first. Another approach is to determine whether the task will complete by the deadline using a history of host availability from the previous day, and then to randomly choose a task that is predicted to complete by the deadline. The author finds that a combined approach of scheduling the task that is expected to complete with the closest deadline is the best method. Although the platform model in that study considers shared and volatile hosts, the platform model assumes that the hosts have identical clock rates and that the platform supports check-pointing. So, the study did not determine impact of relatively slow hosts or task failures on execution for a set of tasks; likewise, the author did not study the effect of resource prioritization (e.g., according to clock rates) or resource exclusion. Batch mode scheduling In the batch mode scheduling, the scheduling approaches are typically more complicated with respect to the previous situation. As a matter of fact, the scheduler, besides the set of resources, knows the set of tasks and it can consider any combination of task/resource. Unfortunately, considering all possible combinations in order to decide the best set of assignments is a wellknown NP-complete problem if throughput is the optimization criterion . For this reason, the batch mode scheduling algorithms proposed in the scienti_c literature provides suboptimal solutions such as Min-Min, Max-min and Su_erage (all described in detail in).
The most relevant batch mode scheduling policy are based on the classic algorithm cited above. For example, Xsufferage is an extension of Sufferage policy that is able to exploit file locality issues without any apriori analysis of the task-file dependence pattern. The idea is that if a file required by some task is already present at a remote cluster, that task would "suffer" if not assigned to a host in that cluster. The Sufferage's value would then be a simple way of capturing such situations and ensuring maximum file re-use.
This is somewhat reminiscent of the idea of task/host affinities introduced in [31], where some hosts are better for some tasks but not for others. The problem of this algorithm regards the necessity to know many information to calculate the completion time of the tasks. In particular, it needs to know the HostSpeed (a measure of the host

speed), the HostLoad (the load of the host due to the local processes) and the TaskSize (the completion time of a task in a host with HostSpeed=1 and HostLoad=0).
Moreover, as we have cited above, some scheduling algorithm needs an extremely fine level of detail such as First-order Prediction-based Dynamic FPLTF (FP Dynamic FPLTF) , abbreviated as FP. FP is a predictionbased scheduling algorithm that works as FPLTF except that it needs the host's latest two load records. The scheduler uses these two records to reconstruct an approximated hosts loading model to predict the hosts's speed in the future based on the linear function. FP is able to achieve good performances but, it needs a level of detail about information of tasks and hosts that is overlay unrealistic to obtain in a Computational Grid, for the reasons previously discussed.

3. Fault-tolerant scheduling

The scheduling approaches discussed above do not consider the occurrence of faults, but as we have just observed, Grid environments are prone of failures. Task failures near the end of the application, and unpredictably slow hosts can cause major delays in application execution. Many solutions propose the replication of the tasks on multiple hosts, either to reduce the probability of task failure or to schedule the application on a faster host. Replication a task may increase the chance that at least one task instance will be completed. One of these proposals is WorkQueue with Replication (WQR): a knowledge-free scheduling algorithm that adds task replication to the classical Workqueue(WQ) scheduler. The beginning of the algorithm execution is the same as the simple Workqueue and continues the same until the bag of tasks becomes empty. At this time, in the simple Workqueue, hosts that _nish their tasks would become idle during the rest of the application execution. Using the replication approach, these hosts are assigned to execute replicas of tasks that are still running. Tasks are replicated until a prede_ned maximum number of replicas is achieved. When a task replica _nishes, its other replicas are canceled. This policy has the drawback of wasting CPU cycles (due to the replicas that do not contribute to the completion of the tasks), which could be a problem if the Desktop Grid is to be used by more than one application. Conversely of these knowledge-free schedulers, the scientific literature has proposed fault-tolerant schedulers that combines replication and knowledgebased scheduling. For example,

Distributed Fault-Tolerant Scheduling (DFTS) is an on-line scheduling policy that uses job replication strategy and it considers available suffcient information to estimate the run-time of the task. In particular, when a job arrives, DFTS chooses a set of n candidates sites for job execution and orders them for an estimate of the job completion time. If the desirable replication threshold is greater than n, DFTS reserves a set of resources equal to the number of unscheduled job replicas. If a job successfully completes, DFTS sends releases message to each site it had reserved such that these sites can be used for running other jobs. The experimental results of DFTS show that performances degrade gracefully in the presence of failures, but it does not consider the amount of waste cycles due to the useless replicas.

Besides the completion time and the number of the replicas desired, other scheduling policies speci_ed also a minimum replication threshold. For example, in the Fault Tolerant Scheduling algorithm (FTSA), along with the job, two values are specified by the user: the number of desired replicas n and the replication threshold k, where k _ n. FTSA picks the n best hosts (ordered by an estimate of the job completion time) and send them the replicas of the task. The system must ensure that k replicas are running. That is, the number of replicas may fall below n due to replica failure, but not below k. Although FTSA could achieve good performance even in presence of fault, it is not clear explained how the user should specify the values of n and k.

Finally, task checkpointing is another means of dealing with task failures since the task state can be stored periodically either on the local disk or on a remote checkpointing server; in the event that a failure occurs, the application can be restarted from the last checkpoint. In combination with checkpointing, process migration can be used to deal with CPU unavailability or when a "better" host becomes available by moving the process to another machine. For example, the EXCL-PRED-CHKPT is a scheduling policy that assumes a checkpoint frequency equals to 2.5 minutes (interval between two checkpoints) and the cost of checkpointing is 15 seconds (time to save/retrieve a checkpoint to/from a remote host). In this work, the authors note that the poor performance of EXCL-PRED-CHKPT is due to the fact that a task is not reassigned when it is assigned to a slow host or when the host becomes unavailable for task execution. Moreover, the authors consider that remote checkpointing or process migration is most likely infeasible in Internet environments, as the application can often consume hundreds of megabytes of memory and bandwidth over the Internet is often limited.

4. Economic models

The scheduling policies previously described consider the resources in Computational Grid freely usable without any constraints. Unfortunately, if constraints are not put on how the resource on how resources are used, they may be misused. For instance, if all resources can be used without any limitation, situation in which all task are submitted to the best resources (e.g., the resource that has the best performance and availability) may arise, thus leaving idle other resources. Moreover, antisocial behaviors, where a user submits a replica of its application on all the resources, with the aim of speculatively exploit them to reduce the execution time of his application, may arise as well. In order to avoid these phenomena, a possible solution consists in associating a cost to each resource, and a budget to each user/application. When an application is run on a resource, a proper amount of (virtual) money is allocated to that application, and is subtracted from the total budget to account for the usage of the resource. The so-called economic approach to resource management has received great attention in the recent Grid literature, where both scheduling algorithms taking into account resource costs and application budgets , and automatic approaches for the computation of resource prices , have been proposed.

Some of the commonly used economic models that can be employed for managing resources environment, include: the commodity market model, the posted price model, the bargaining model, the tendering/contract-net model, the auction model, the bid-based proportional resource sharing model, the community/coalition/bartering model and the monopoly and oligopoly. One example of commodity market model is the Nimrod/G resource broker: a global resource management and scheduling system shown on Figure 2, that supports deadline and economy-based computations in Grid computing environments for parameter sweep applications. In particular, in the Nimrod/G application level resource broker, three adaptive algorithms for scheduling are incorporated: (i) time minimization, (ii) cost minimization and (iii) none minimization. The time minimization algorithm attempts to complete an experiment as quickly as possible, within the available budget.

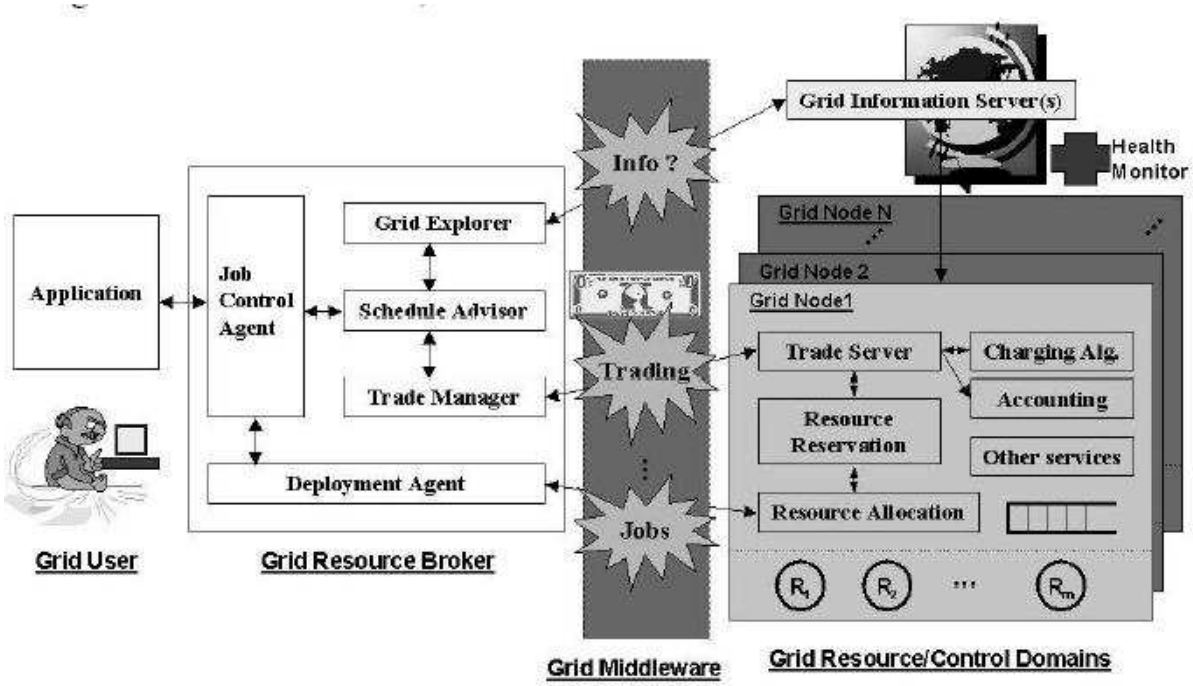

Figure.2 Nimrod/G resource broker

The cost minimization algorithm attempts to complete an experiment with the lowest cost as possible within the deadline. Finally, the none minimization attempts to complete the experiment within the deadline and cost constraints without minimizing either. The price of each resource has been Figure 2: Nimrod/G resource broker established dynamically using GRid Achitecture for Computational Economy(GRACE) but the results discussed are based on an arbitrary assignment for demonstrate purpose only. Thus, it is di_cult to evaluate the effective performance of this scheduling algorithm. Also the researchers of the University of California at Berkeley propose a market-based approach to cluster resource management based on the notion of a Computational Economy. In this model, resource rights for a shared resource are encapsulated as tickets. A resource is represented by a total of T tickets. An application holding t tickets competing for use of that resource obtains a share of t=T of the resource. The main contribution of this work is the discussion about the systems that implements marketbased ideas to cluster resource management. The models studied (Ferguson, Spawn, Popcorn, Mariposa, SC centers) apply di_erent interpretation of tokens distribution and what resources share, but the main observation regards that most of the work so far has focused mainly on the economic front-end and layer. Very little attention has been paid to the end-to-end problem:

An approach that is not based on the price of the resources presents a baseline architecture for bootstrapping economies based on peer-to-peer bartering. Bartering strategies specify how peers negotiate exchange rates for peering and how peers execute the peering protocol. Negotiating exchange rates involves determining what amount of resources a peer X exchanges with a peer Y as part of the peering and how many such exchanges will occur. One simple strategy based on reciprocity that has proven to be remarkably robust and e_ective against a wide range of competing strategies is TIT FOR TAT . TIT FOR TAT is thestrategy of beginning with cooperation and, thereafter, doing whatever the other peer did in the previous round. It is simple, encourages cooperation, punishes defection (but is forgiving), and in practice outperforms virtually all competing strategies in a number of situations. Unfortunately, the work explains the theoretical aspects that should guaranteed good performance of the model proposed, but no results are presented to evaluate this bartering model.
.

VI Analysis

1. Knowledge-base algorithm versus knowledge-free algorithm

Obtain information about the status of the resource is often difficult, especially in Grid environments. This is due to some characteristics that are intrinsic to Grids such as heterogeneity and volatility of the resources. Moreover, in order to achieve good performance, it is necessary to know the status of the resources in the next future, because just monitoring the resource (when it is possible) is not enough. The scientific literature has proposed tools able to obtain dynamic information such as host load and network bandwidth with some good results. Unfortunately these are only initial encouraging results, thus the knowledge-free scheduling algorithms are still popular in Grid environments. On-line scheduling versus batch mode scheduling The main di_erence, in terms of performance, between on-line scheduling and batch mode scheduling regards principally the arrival rate. When the arrival rate is low, machines may be ready to execute a task as soon as it arrives at the scheduler. Therefore, it may be bene_cial to use the scheduler in the on-line mode so that a task does not need to wait

the next scheduling event to begin its execution. In batch mode, the scheduler considers a bag of tasks for matching and scheduling at each scheduling event. This enables the scheduling algorithm to possibly make better decisions, because the scheduler have the resource requirement information for all tasks, and know about the actual execution times of a larger number of tasks (as more tasks might complete while waiting for the scheduling event). When the task arrival rate is high, there will be a su_cient number of tasks to keep the machines busy in between the scheduling events, and while an assignment is being computed.

2. Fault-tolerant scheduling versus no fault-tolerant scheduling

Although scheduling and fault tolerance have been traditionally considered independently from each other, there is a strong correlation between them. As a matter of fact, each time a fault-tolerance action must be performed, i.e. a replica must be created or a checkpointed job must be restarted, scheduling decision must be taken in order to decide where these jobs must be run, and when their execution must start. A scheduling decision taken by considering only the needs of the faulty task may thus strongly adversely impact non-faulty jobs, and vice versa. Therefore, scheduling and fault tolerance should be jointly addressed in order to simultaneously achieve fault tolerance and satisfactory performance.

VII Conclusions

The intention of this chapter has been to provide the related work in the area of resource management. This has been done through the presentation of taxonomy on the scheduling policies used in Grid computing. From our study, we can assert that usually the scheduling literature has considered efficiency and robustness as orthogonal aspects, that is their interactions are not taken into account when scheduling applications on Computational Grid. Unfortunately, as already mentioned before, in Desktop Grids faults may occur, and in this case the execution time of the applications gets much larger, as there is the need to recover from the fault. Consequently, there is the need of exploring scheduling strategies that attempts to maximize application performance in face of occurrence of faults.

For this reason, in the next chapters of this thesis, we propose a novel fault-tolerant and knowledge-free scheduler based on the

WQR algorithm able to achieve performance better than alternative scheduling strategies.

Bibliography

1) David P. Anderson. BOINC: A System for Public-Resource Computing and Storage. Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing, 2004.

2) D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D.Werthimer. SETI@home: An experiment in public resource computing. Communications of the ACM, Nov. 2002, Vol. 45 No. 11, pp. 56-61.

3) G. Fedak and C. Germain and V. N'eri and F. Cappello XtremWeb: A Generic Global Computing System. In IEEE Int. Symp. on Cluster Computing and the Grid, 2001.

4) N. Andrade, R. Santos, A. Andrade, R. Novaes, M. Mowbray, W. Cirne, and F.V. Brasileiro. Labs of the World, Unite!!!. In , 2005.

5) B. Dragovic and K. Fraser and S. Hand and T. Harris and A. Ho and I. Pratt and A.War_eld and P. Barham and R. Neugebauer Xen and the Art of virtualization. In Proceedings of the ACM Symposium on Operating Systems Principles, 2003.

6) Elizeu Santos-Neto and Walfredo Cirne and Francisco Brasileiro and Aliandro Lima Exploiting Replication and Data Reuse to E_ciently Schedule Data-Intensive Applications on Grids. In 10th JSSPP, 2004.

7) D.P. da Silva, W. Cirne, and F.V. Brasileiro. Trading Cycles for Information: Using Replication to Schedule Bag-of-Tasks Applications on Computational Grids. In Proc. of EuroPar 2003, volume 2790 of Lecture Notes in Computer Science, 2003.

8) H. Casanova, A. Legrand, and D. Zagorodnov et al. Heuristics for Scheduling Parameter Sweeping Application in Grid Environments. In Proc. of Heterogeneous Computing Workshop, IEEE CS Press, 2000.

9) M. Maheswaran and H.J. Siegel A Dynamic Matching and Scheduling Algorithm for Heterogeneous Computing Systems. in Proceedings of the Seventh Heterogeneous computing Workshop HCW '98, 1998.

10) J. H. Abawajy Fault-Tolerant Scheduling Policy for Grid Computing Systems. in Proceedings of the 18th International Parallel and Distribuited Processing Symposium (IPDPS'04), 2004.

11) J. Weissman and D. Womack. Fault Tolerant Scheduling in Distributed Networks. Technical Report TR CS-96-10, Department of Computer Science, University of Texas, San Antonio, Sept. 1996.

12) Derrick Kondo, Andrew A. Chien, Henri Casanova Resource Management for Rapid Application Turnaround on Enterprise Desktop Grids. Proceedings of Super Computing Conference, 2004.

13) R. Buyya and M. Murshed GridSim: A Toolkit for the Modeling and Simulation of Distributed Resource Management and Scheduling for Grid Computing. in the Journal of the Concurrency and Computation: Practice and Experience,vol. 14, no. 13-15, pp. 1175-1220, Wiley Press, USA, 2002.

14) B.Chun and D.Culler Market-Based Proportional Resource Sharing for Clusters. Technical Report CSD1092, University of California at Berkeley, January 2000.

15) R. Buyya, J. Giddy, and D. Abramson An Evaluation of Economybased Resource Trading and Scheduling on Computational Power Grids for Parameter Sweep Applications. Prooceedings of The Second Workshop on Active Middleware Services (AMS 2000), 2000.

16) B.Chun, Y.Fu, A.Vahdat Bootstrapping a Distributed Computational Economy with Peer-to-Peer Bartering. Workshop on Economics of Peerto-Peer Systems, 2003

17) P. Francis, S. Jamin, V. Paxson, L. Zhang and D.F. Gryniewicz and Y. Jin An Architecture for a Global Internet Host Distance Estimation Service. In the proocedings of IEEE INFOCOM, 1999.

18) B. Lowekamp, N. Miller, T. Gross, P. Steenkiste, J. Subhlok and D. Sutherland A resource query interface for network-aware applications. In the Journal of Cluster Computing, 1999.

19) R. Wolski Dynamically forecasting network performance using the network weather service. In the Journal of Cluster Computing, vol.1, no.1, pp.119-132, 1998.

20) G. Woltman The great internet mersenne prime search