A Study of Deadline Scheduling for Client-Server Systems on the Computational Grid

Atsuko Takefusa
Japan Society for the Promotion of Science /
Tokyo Institute of Technology
2-12-1 Tokyo 152-8552, Japan
takefusa@is.titech.ac.jp

Henri Casanova
University of California, San Diego
9500 Gilman Dr. La Jolla, CA 92093-0114, USA
casanova@cs.ucsd.edu

Satoshi Matsuoka
Tokyo Institute of Technology / JST
2-12-1 Tokyo 152-8552, Japan
matsu@is.titech.ac.jp

Francine Berman
University of California, San Diego /
San Diego Supercomputer Center
9500 Gilman Dr. La Jolla, CA 92093-0114, USA
berman@sdsc.edu

Abstract

The Computational Grid is a promising platform for the deployment of various high-performance computing applications. A number of projects have addressed the idea of software as a service *on the network. These systems usually implement client-server architectures with many servers running on distributed Grid resources and have commonly been referred to as* Network-enabled servers *(NES). An important question is that of* scheduling *in this multi-client multi-server scenario. Note that in this context most requests are computationally intensive as they are generated by high-performance computing applications. The Bricks simulation framework has been developed and extensively used to evaluate scheduling strategies for NES systems. In this paper we first present recent developments and extensions to the Bricks simulation models. We discuss a* deadline scheduling *strategy that is appropriate for the multi-client multi-server case, and augment it with "Load Correction" and "Fallback" mechanisms which could improve the performance of the algorithm. We then give Bricks simulation results. The results show that future NES systems should use deadline-scheduling with multiple fallbacks and it is possible to allow users to make a trade-off between failure-rate and cost by adjusting the level of conservatism of deadline-scheduling algorithms.*

1. Introduction

The emergence of Computational Grid environments [17, 19] has caused much excitement in the high-performance computing (HPC) community. Advances in network technology have made it increasingly possible to deploy HPC applications over the wide-area, thereby accessing unprecedented amounts of compute and storage resources. Many Grid software systems have been developed and it has become possible to deploy real applications on these systems [32, 16, 21, 23]. A crucial issue for the efficient deployment of HPC applications on the Grid is that of *scheduling* [6]. Most scheduling works addressing Grid environments aim at improving execution time (or *makespan*) of a single application executed on behalf of a single user [8, 7, 14, 20, 38, 33].

A number of projects pre-dating the advent of the Grid have addressed the idea of software as a *service* on the network. These systems have been traditionally called *Network-enabled Servers* (NES) [13, 32, 22, 15] and are currently in use for many types of applications. These systems usually implement client-server architectures and provide users with an RPC-style programming model. Users can then easily access (interactively or via programs) computational modules and compute cycles on remote resources. Many high-profile applications from science and engineering are amenable to this programming model [36, 1, 25, 5, 18, 28, 34, 4, 31]. Each application consists of a large number of more or less *independent* tasks. Despite their simple structure, these applications require tremendous amounts of computational power. A recent effort as part of the Global Grid Forum [19] aims at making recommendations for implementing a middleware layer that supports these applications [26].

An important issue is then the question of scheduling in this multi-client multi-server scenario where several applications compete for the access to NES resources. One promising approach to the problem is that of a *resource economy* model [40, 30, 10]. The expectation is that as the Grid becomes more pervasive, the notion of a *Grid currency* will allow resources owners to "charge" for resource usage. At this time, these works are mostly speculative and no actual economical model is in practical use or agreed-upon.

The work in [2] presents a study of deadline-scheduling algorithms for a particular economy model. Deadline scheduling is clearly the right strategy in our multi-client scenario: users specify deadlines for the tasks of their applications and can "spend" more to get tighter deadlines (as seen in [2]).

Given the current lack of consensus on Grid economy models, we take a general approach: we assume that each request comes with a deadline requirement and our goal is to minimize the *overall* occurrences of deadline misses as well as their magnitude. Our cost-model assumes that resources are priced proportionally to their performance (e.g. $ per MFlops). When a more sophisticated Grid economy model becomes prominent, we will extend this work accordingly. As a first step towards deriving an effective scheduling algorithm that satisfies such a requirement, we propose a simple deadline scheduling strategy for NES systems. We propose various parametrizations and strategies that we conjecture would improve the overall scheduling results, and use the Bricks simulation framework [37, 3, 9] for evaluation purposes.

More specifically, we first describe improvements to Bricks that allow for more scalable and realistic simulations of Grid environments. We then give a simple deadline scheduling algorithm which aims at minimizing deadline misses, and then augment it with "Load Correction" and "Fallback" mechanisms which could improve the performance of the algorithm in our context. We focus on multi-user scenarios and investigate the relationship between resource load, resource cost, conservatism of performance prediction, and the efficacy of several variants of our deadline scheduling strategy. We present results from thousands of simulation runs, obtained by massive parallel parameter-sweep running of Bricks over a large cluster of Linux PCs, using a recently available Grid middleware product (APST [14]), using the cluster nodes as Grid compute resources. The results show that future NES systems should use deadline-scheduling with multiple fallbacks and it is possible to allow users to make a trade-off between failure-rate and cost by adjusting the level of conservatism of deadline-scheduling algorithms.

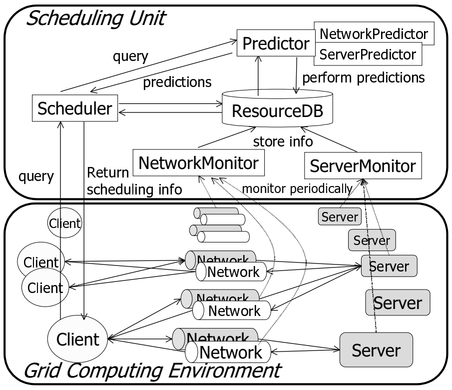

Figure 1. The Bricks Architecture.

2. The Bricks System and the Extension

Bricks is a simulation framework to evaluate the performance of client-server systems in Grid environments. It is a discrete event simulator written in Java and consists of a simulated Grid Computing Environment, and of a Scheduling Unit (see Figure 1).

The Grid Computing Environment provides the following set of simulation components:

- Simulated Computational Grid topology,
- Various resource behavioral models (e.g. load traces, queues),
- Client model (e.g. request arrival times, etc.).

The Scheduling Unit consists of the following canonical Grid scheduling modules [6]:

- Scheduler
- Predictor
- Resource monitor
- Resource database (DB)

As for the Grid Computing Environment, the components of the Scheduling Unit are written in Java. We have demonstrated in previous work [37] that it is easy to "plug in" various external components into the Scheduling Unit (e.g. predictor from the Network Weather Service [39]). We refer the reader to [3, 37] for evaluation and validation results concerning Bricks.

A problem with the first version of Bricks had been its scalability; In order to investigate characteristics of a variety of scheduling algorithm in a more scalable, realistic Grid

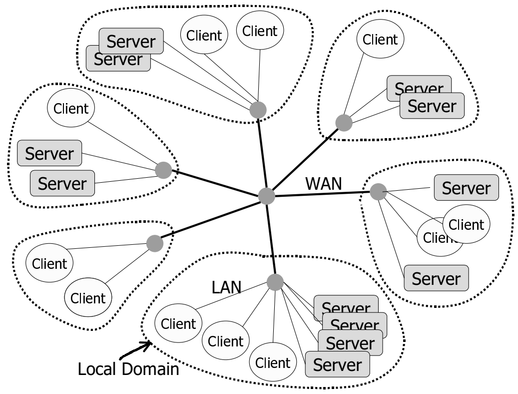

Figure 2. An Example of Network Topology.

setting, we extended Bricks to incorporate the scalable network model shown in Figure 2. We extended Bricks to support tree-based network structures. Each network between a client and a server consists of intermediate nodes and network queues which represent LANs and WANs. This extension allows the use of Internet topology generators such as GT-ITM [11], tiers [11], and BRITE [27]. These generators produce tree-structured topologies with LAN and WAN links with fixed bandwidth.

2.1. Simulation of *Background* Load

In this section, we describe the simulation of background load on resources (servers and network links). This load is generated by applications and users outside our system (and are not known to our scheduler) and in practice what would be experienced on a real Grid. In Bricks, resources are represented with *queues* and background load can be simulated with one of the following two models:

Extraneous data/job model Background resource congestion is modeled by artificially injecting external requests into the system. Resources have constant performance characteristics [3].

Trace model In this model, the performance characteristics of resources (e.g. CPU load) are modeled after *traces* (vectors of time-stamps values). Typically, those traces come from measurements on real resources [37] (e.g. with the NWS [39]).

In the extraneous data/job model, one needs to specify only several parameters before the simulation. However, it is difficult to control the variance of the congestion, especially for network queues. In the trace model, the simulation cost is lower than in the former model. Note that Grid topology configurations require large amounts of trace data if the trace model is to be used for all resources.

All the simulation results in this paper use the extraneous job model for servers (with Poisson inter-arrival time) and the trace model for network links. We took advantage of Paxson's high-quality self-similar network traffic traces [29] for all network simulations. This method makes it possible to generate a variety of realistic network traffic with a relatively small set of parameters as input. We did not have access to a wide variety of high-quality traces for modeling server load. Therefore, we opted for the extraneous job model and generated series of random extraneous requests.

3. Deadline Scheduling

Currently deployed NES systems (NetSolve [13] and Ninf [32]) use the greedy *on-line* scheduling algorithm described in [24] as MCT (Minimum completion Time). For each incoming request, the scheduler assigns the request to the server that completes it the earliest. This is an aggressive approach which does not take into account possible deadline requirements from the user. In the rest of the paper we call this algorithm *Greedy* . As NES systems become more widely deployed on the Computational Grid, it becomes necessary to fully support the notion of deadline. Deadline scheduling is a strategy which aims at meeting user-supplied job deadline specifications. We first show a simple deadline scheduling algorithm. We then propose two mechanisms that improve the performance of the algorithm in our context.

3.1. A Deadline Scheduling Algorithm

We propose an on-line scheduling algorithm with the goal of minimizing the number of missed deadlines. The algorithm is invoked at every job arrival:

1. Compute $T_{until\ deadline}$ as
$$T_{until\ deadline} = T_{deadline} - now \qquad (1)$$
where $T_{deadline}$ is the job's deadline specification, and now is the current time.

2. Estimate the job processing time T_{S_i} on each available server S_i ($0 < i \leq n$).
$$T_{S_i} = W_{send}/P_{send} + W_{recv}/P_{recv} + W_s/P_{serv} \qquad (2)$$
where W_{send}, W_{recv}, and W_s denote to the data sizes transmitted from a client to a server and from the server to the client and logical computation "cost" (in some

arbitrary units) of the job. $P_{send}, P_{recv}, P_{serv}$ denote predicted network throughputs from the client to the server, from the server to the client, and the server performance (in units per second), respectively.

3. Compute the target processing time T_{target} as:

$$\begin{aligned} T_{target} = \quad & T_{until\ deadline} \times Opt \\ \text{with} 0 < Opt \leq 1 & \end{aligned} \quad (3)$$

Opt is a parameter that denotes the expected accuracy of performance prediction in the simulated Grid environment. In general, schedulers use prediction concerning the performance of Grid resources. Such predictions are available through systems such as the NWS [39] and contain some error.

Opt allows to specify how *conservative* the algorithm is. Setting *Opt* to 1 means that one has high confidence in the predictions. Smaller values account for expected errors in performance prediction. The rationale is that if *Opt* is low, then scheduling should be more conservative and select resources that might be faster than what is needed, at a higher cost. For instance, setting *Opt* to 0.5 means that the algorithm will attempt to have tasks complete "twice sooner" than their deadlines. Future work will address automatic tuning of *Opt*. For instance, the scheduler could maintain a history of observed prediction errors and use that history to dynamically increasing or decreasing the value of *Opt*.

4. select a suitable server S_i whose T_{S_i} is not over and nearest to T_{target}:

$$\begin{aligned} Diff \quad & = T_{target} - T_{S_i} \geq 0 \\ & Diff \text{ is smallest.} \end{aligned} \quad (4)$$

However, if there are not any servers S_i which satisfy $Diff \geq 0$, then select a server with the smallest $|Diff|$.

3.2. A Load Correction Mechanism

We have mentioned that the scheduler uses load measurements (or predictions) from deployed services such as the NWS [39]. The algorithm as is it described in the previous section uses such predictions but in fact possesses additional knowledge about the usage of resources. Indeed, previous scheduling decisions might have been made that will lead to an increase in load of a given resource. In addition, monitoring systems do not perceive load changes instantaneously. Let us take an example to illustrate this problem. Assume that we have two resources, one of them twice as fast as the other, and that we have 3 identical tasks to schedule. If the scheduler does not keep track somehow of previous scheduling decisions, all 3 tasks will be sent to the fastest resource. However, the optimal schedule is to schedule 2 tasks on the fastest resource and 1 on the other resource.

A simple way to address that issue is to modify the scheduling algorithm so that it uses *corrected* load values. This strategy has been proposed for on-line schedulers within NES systems. We address this problem for the load of computational resources (servers) and we leave a Load Correction mechanism for network links as future work. The idea is to modify load predictions from the monitoring system, $Load_{S_i}$, as follows to obtain corrected load values:

$$Load_{S_i corrected} = Load_{S_i} + numjobs_{S_i} \times pload \quad (5)$$

where $numjobs_{S_i}$ is the number of jobs that have been scheduled (and have not completed) by our scheduler, and $pload$ is an arbitrary value that determines the magnitude of the correction. The results in this paper use $pload = 1$. The rationale behind that choice is that CPU-bound tasks add 1 to the CPU load (e.g. the load of a server running three CPU-bound task should be 3.0. For now, we assume that our applications consist of CPU-bound tasks.)

3.3. A Fallback Mechanism

The scheduling algorithm uses estimations (predictions) for data transfer times and computation times for each request. In certain cases, it may be possible that once the input data has reached the server that server is actually unable to complete the task by its deadline. This may be due to errors in the estimation of input data transfer times. Furthermore, we assume that the servers are used in a First Comes First Served fashion (FCFS). Due to performance prediction errors, it may be the case that tasks are *out-of-order* in a server's queue, with respect to what the scheduler intended. A solution to that problem is then to push some of the scheduler's functionalities to the servers themselves.

Once input data for a task has reached a server, that server can estimate whether it will be able to complete the task by the required deadline. Indeed, a server has knowledge about the current state of its queue. If the server determines that the task will not meet the deadline, then it can notify the corresponding client that will then re-submit the task to the system. Such notification-resubmission mechanism is called the *fallback*. We impose a limit on the number of times a task can benefit from the Fallback mechanism.

The conditions under which a task *fallbacks* are as follows:

$$T_{until\ deadline} < T_{send} + ET_{exec} + ET_{recv} \quad (6)$$

$$N_{fallbacks} \leq N_{max.\ fallbacks}. \quad (7)$$

where $T_{until\ deadline}$ is determined in Eq. (1). T_{send} is the observed duration of the data transfer from the client to the server. ET_{recv} is an estimation of the duration of the output data transfer. ET_{exec} denotes the job processing time estimated by the server at the input data transfer completes. $N_{fallbacks}$ is the number of fallbacks that the task has already gone through, and $N_{max.\ fallbacks}$ is the maximum number of fallbacks allowed. Of course, the original deadline is used for resubmissions and the scheduler repeatedly processes the step 1 to 4 in Section 3.1 for each submission.

The simulation results in this paper impose limits of 1 to 5 fallbacks for each task. Note that it is possible for a server to decide to still execute a task if its deadline it not "too overdue". One could define a threshold beyond which a task must fallback.

4. Experiments

We now describe the performance of the *Deadline* scheduling algorithm using the extended Bricks system. Using the Presto II cluster (Dual Pentium III 800MHz, × 64 nodes) at the Tokyo Institute of Technology, we performed approximately 2,500 runs of Bricks simulations varying the algorithm, client, server, network topology, *Opt*, and the load of the system. One simulation takes about 30-60 minutes for simulating 75 nodes for 24 hours, using the Java 2 Runtime Environment (1.3.0) and the Java HotSpot Client VM. Simulations were run using the APST software [14] to easily deploy and schedule the various experiments in a parallel parameter-sweep fashion over the available cluster resources.

4.1. Simulated Environment

The goal of our simulation is twofold. First, we wish to compare the *Greedy* algorithm to our *Deadline* scheduling. Second, we study the impact of the *Opt* parameter, the Load Correction and Fallback mechanisms on the numbers of deadline misses and experienced resource costs.

Features of the simulated environment are shown in Table 1. Using *Grid Computing Environment* parameters we generated 5 different environments. All experimental results are averaged over those 5 environments. LAN/WAN throughput during the simulations are determined by self-similar traces (Section 2.1) whose standard deviation is 10% of the average throughput.

The *Client Job Characteristics* in Table 1 are generated during the simulations. In order to simulate a mixed set of jobs, the number of logical job instructions is set to 1.5-1080 [Gops] making the processing time 5-60 [min] for the average unloaded server performance (300 [Mops/sec]). The average request inter-arrival time, *Int*, allows to study high-workload (*Int* = 60), medium-workload (*Int* = 90), and low-workload (*Int* = 120) scenarios.

The deadline specification for each job is simulated as follows. For each job, we compute the time to completion on a hypothetical "average" unloaded server. We then scale that time by a factor, *DF* , that is uniformly distributed over the interval 1-3. This allows us to generate a mix of jobs with various deadline specifications.

4.2. Simulation Results

Figure 3 shows the *Failure Rate* for the *Greedy* algorithm and of the *Deadline* algorithm with different values of *Opt* (denoted by D-*Opt* on the x axis). The Failure Rate is defined as the percentage of requests that missed their deadlines. Results are shown for the three different workload conditions mentioned in Section 4.1. Four versions of each scheduling algorithm were used: with or without the Load Correction mechanism, and with or without the Fallback mechanism. This is denoted on the graphs in Figure 3 by x/x, L/x, x/F, and L/F where 'L' is Load Correction, 'F' is Fallback, and 'x' is *not used*. These experiments use a maximum number of fallbacks equal to 1. With this notation, the *Greedy* algorithm is x/x (no Load Correction and no Fallback). In the three workload conditions one can make the following two observations: (i) the Fallback mechanism leads to dramatic reductions of the failure rate; (ii) the Load Correction mechanism leads only to marginal failure rate improvements. Observation (ii) is due to inaccuracies in resource performance predictions. Jobs invoked by users outside our system cause the resource utilization to change unexpectedly and the same is true for communication delays as we model network links with traces (see Section 2.1). Therefore, according to our results, the Load Correction mechanism may not be useful for NES systems in scenario with a large numbers of clients and servers.

Of course, the *Deadline* algorithm leads to better failure rates when *Opt* is small (since the algorithm tries to enforces stricter deadline than what is necessary). This must be put in perspective by looking at the resource costs, however.

Figure 4 shows the average resource *Cost* over all requests for the same simulation runs as in Figure 3. Here again one can make two observations: (i) the *Greedy* algorithm leads to high costs than the *Deadline* algorithm; (ii) costs decrease when *Opt* increases as the algorithm becomes less conservative. The main result here is that even a conservative deadline scheduling algorithm is preferable to what is currently being implemented within NES systems. Furthermore, using the results in Figures 3 and 4, it is possible to make a "trade-off" between failure-rate and cost by adjusting the level of conservatism of the *Deadline* algorithm.

Figures 5 and 6 are similar to the previous two Figures but show experiments with no Load Correction and different values for the maximum number of allowed fallbacks

Table 1. Parameters used in the experiments.

Grid Computing Environment		
# of local domains	10	Fixed
# of local domain nodes	5-10	Uniform distribution
Ratio of clients and servers	1:1	Fixed
Average LAN throughput	50-100[Mbits/s]	Uniform distribution
Average WAN throughput	500-1000[Mbits/s]	Uniform distribution
Server performance	100-500[Mops/s]	Uniform distribution
Average server load	0.1	Fixed
# of extraneous job instructions	50[Mops]	Fixed
Logical packet size	10[Mbits]	Fixed
Client Job Characteristics		
Transmitted data size (send/recv)	100-5000[Mbits]	Uniform distribution
# of job instructions	1.5-1080[Gops]	Uniform distribution
Interval of job invocations (*Int*)	60/90/120[min]	Poisson arrivals
Deadline factor (*DF*)	1.0-3.0	Uniform distribution
Scheduling Unit		
Scheduling Algorithm	*Deadline* /*Greedy*	
Opt	0.5/0.6/0.7/0.8/0.9	
Load Correction mechanism	on / off	
Fallback mechanism	$N_{max.\ fallbacks}$ = 0/1/2/3/4/5	

(0 to 5). From Figure 5 one can see that much improvement can be gained by allowing for multiple fallbacks in all three workload scenarios, for all the scheduling algorithms. Figure 6 shows that allowing for multiple fallbacks leads to small increases in resource costs. The main result here is that schedulers which are part of NES systems should allow for multiple fallbacks as part of their standard mechanisms.

5. Related Work

Nimrod [2] is a Grid system for parameter sweep applications and uses a self-scheduler that allocates tasks into lower priority (and performance) servers at first, collects execution results for the next scheduling, and then makes all tasks meet the application deadline. Experiments in actual environments (see [2]) showed that the scheduler tends to select high priority resources as the deadlines get closer. By contrast to our work, scheduling in Nimrod targets parameter sweep applications from a single user.

Also, there have been several Grid performance evaluation systems. MicroGrid [35] emulates a virtual Globus [16] Grid on cluster of actual machines. MicroGrid software is not publicly available at the time this paper is being written and it will not be usable for running very large number of long-term experiments. Indeed, MicroGrid is an emulator and runs actual application code, which would lead to prohibitive simulation times in our context. Simgrid [12] is a trace-based discrete event simulation toolkit and provides primitives for simulation of application scheduling in Grid environments. At this time, Simgrid does not provide the network-modeling features necessary to run simulations in topologies such as the one in Figure 2.

6. Conclusions and Future Work

We proposed a deadline scheduling algorithm and Load Correction and Fallback mechanisms that improve the algorithm's efficacy. We investigated its performance in multi-client multi-server scenarios with the improved Bricks simulation framework. The experiments showed:

- A *Greedy* algorithm leads to higher costs than *Deadline*.
- Conservative deadline estimation (small values of *Opt*) results in less failure rates at the expense of resource costs.
- The Load Correction mechanism may not be useful pending further simulations.
- The Fallback mechanism leads to drastic reductions of failure rates and multiple fallbacks do not lead to much cost increase.

The results suggest that future NES systems on the Grid use deadline-scheduling with multiple fallbacks as multiple users will compete for resources. Furthermore, the results show that it is possible to allow users to make a trade-off between failure-rate and cost by adjusting the level of conservatism of deadline-scheduling algorithms. The current NES APIs do not support such a feature as no Grid resource economy model is currently enforced. However, this work will become emminently applicable as soon as some consensus concerning resource economy is reached.

For future work, we plan to make Bricks support more sophisticated economy models as they become available, investigate their feasibility, and then, implement the deadline scheduling algorithm within actual NES systems (starting with Ninf [32]).

Acknowledgments

We would like to thank all the members of the Matsuoka Laboratory at Tokyo Institute of Technology for their great support with respect to conducting the experiments on the PC cluster.

This research was partially supported by the Ministry of Education, Science, Sports and Culture, Grant-in-Aid for JSPS Fellows, 7724, 2000 and the JST PRESTO program.

References

[1] D. Abramson, M. Cope, and R. McKenzie. Modeling Photochemical Pollution using Parallel and Distributed Computing Platforms. In *Proceedings of PARLE-94*, pages 478–489, 1994.

[2] D. Abramson, J. Giddy, and L. Kotler. High Performance Parametric Modeling with Nimrod/G: Killer Application for the Glocal Grid? In *Proceedings of IPDPS2000*, 2000.

[3] K. Aida, A. Takefusa, H. Nakada, S. Matsuoka, S. Sekiguchi, and U. Nagashima. Performance Evaluation Model for Scheduling in Global Computing Systems. *International Journal of High-Performance Computing Applications*, 14(3):268–279, 2000.

[4] A. Amsden, J. Ramshaw, P. O'Rourke, and J. Dukiwica. Kiva: A computer program for 2- and 3-dimensional fluid flows with chemical reactions and fuel sprays. Technical Report LA-10245-MS, Los Alamos National Laboratory, 1985.

[5] J. Basney, M. Livny, and P. Mazzanti. Harnessing the Capacity of Computational Grids for High Energy Physics. In *Conference on Computing in High Energy and Nuclear Physics*, 2000.

[6] F. Berman. *The Grid, Blueprint for a New computing Infrastructure*, chapter 12. Morgan Kaufmann Publishers, Inc., 1998. Edited by Ian Foster and Carl Kesselman.

[7] F. Berman, H. Casanova, D. Zagorodnov, and A. Legrand. Using Simulation to Evaluate Scheduling Heuristics for a Class of Applecations in Grid Environments. CS N^o 99-46, École Normale Supérieure de Lyon, 1999.

[8] F. Berman, R. Wolski, S. Figueira, J. Schopf, and G. Shao. Application-Level Scheduling on Distributed Heterogeneous Networks. In *Proceedings of the 1996 ACM/IEEE Supercomputing Conference*, 1996.

[9] Bricks. `http://ninf.is.titech.ac.jp/bricks/`.

[10] R. Buyya, D. Abramson, and J. Giddy. An Economy Driven Resource Management Architecture for Global Computational Power Grids. In *Proceedings of The 2000 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 2000)*, June 2000.

[11] K. L. Calvert, M. B. Doar, and E. W. Zegura. Modeling Internet Topology. In *IEEE Communications*, 1997.

[12] H. Casanova. Simgrid: A Toolkit for the Simulation of Grid Application Scheduling. In *CCGRID 2001*, 2001.

[13] H. Casanova and J. Dongara. NetSolve: A Network Server for Solving Computational Science Problems. In *Proceedings of the 1996 ACM/IEEE Supercomputing Conference*, 1996.

[14] H. Casanova, G. Obertelli, F. Berman, and R. Wolski. The AppLeS Parameter Sweep Template: User-Level Middleware for the Grid. In *Proceedings of the 2000 ACM/IEEE Supercomputing Conference*, 2000.

[15] J. Czyzyk, M. Mesnier, and J. Moré. NEOS: The Network-Enabled Optimization System. Technical Report MCS-P615-1096, Mathematics and Computer Science Division, Argonne National Laboratory, 1996.

[16] I. Foster and C. Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *International Journal of Supercomputer Applications*, 11(2):115–128, 1997.

[17] I. Foster and C. Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1999.

[18] A. R. Gallant and G. Tauchen. SNP: A Program for Nonparametric Time Series Analysis. Duke economics Working Paper #95-26, Duke University, 1997. v8.6 (revised 1997).

[19] Global Grid Forum. `http://www.gridforum.org/`.

[20] J.-P. Goux, S. Kulkarni, J. Linderoth, and M. Yoder. An Enabling Framework for Master-Worker Applications on the Computational Grid. In *Proceedings of the 9th IEEE International Symposium on High Performance Distributed Computing (HPDC)*, pages 43–50, 1999.

[21] A. S. Grimshaw, W. A. Wulf, J. C. French, A. C. Weaver, and P. F. Reynolds. Legion: The Next Logical Step Toward a Nationwide Virtual Computer. CS 94-21, University of Virginia, 1994.

[22] N. Kapadia, J. Forter, and C. Brodley. Predictive Application-Performance Modeling in a Computational Grid Environment. In *Processings of the 8th IEEE International Symposium on High Performance Distributed Computing (HPDC8)*, 1999.

[23] M. Litzkow, M. Livny, and M. W. Mutka. Condor - A Hunter of Idle Workstations. In *Proceedings of the 8th International Conference on Distributed Computing Systems*, pages 123–130, 1988.

[24] M. Maheswaran, S. Ali, H. Siegel, D. Hensgen, and R. Freund. Dynamic Mapping of a Class of Independent Tasks onto Heterogeneous Computing Systems. *Journal of Parallel and Distributed Computing*, 59:107–131, 1999.

[25] A. Majumdar. Parallel Performance Study of Monte-Carlo Photon Transport Code on Shared-, Distributed-, and Distributed-Shared-Memory Architectures. In *Proceedins of the 14th Parallel and Distributed Processing Symposium, IPDPS'00*, pages 93–99, May 2000.

[26] S. Matsuoka and H. Casanova. Network-Enabled Server Systems and the Computational Grid, 2000. Global Grid Forum White Paper.

[27] A. Medina, I. Matta, and J. Byers. On the Origin of Power Laws in Internet Topologies. *Computer Communication Review*, 30(2):18–28, 2000.

[28] W. Nelson, H. Hirayama, and D. Rogers. The EGS4 Code system. Technical Report SLAC-265, Stanford Linear Accelerator Center, 1985.

[29] V. Paxson. Fast, Approximate Synthesis of Fractional Gaussian Noise for Generating Self-Similar Network Traffic. *Computer Communication Review*, 27(5):5–18, 1997.

[30] J. S. Plank, R. Wolski, J. Brevik, and T. Bryan. G-Commerce: The Study and Building of Computational Economies for the Computational Grid, 2000. Workshop on Clusters and Computational Grids for Scientific Computing
`http://www.cs.utk.edu/ dongarra/lyon2000/lyon-2000.html`.

[31] S. Rogers. A Comparison of Implicit Schemes for the Incompressible Navier-Stokes Equations with Artificial Compressibility. *AIAA Journal*, 33(10), Oct. 1995.

[32] M. Sato, H. Nakada, S. Sekiguchi, S. Matsuoka, U. Nagashima, and H. Takagi. Ninf: A Network based Information Library for a Global World-Wide Computing Infrastracture. In *Proceedings of HPCN'97 (LNCS-1225)*, pages 491–502, 1997.

[33] J. M. Schopf and F. Berman. Stochastic Scheduling. In *Proceedings of the 1999 ACM/IEEE Supercomputing Conference*, 1999.

[34] S. Sciutto. AIRES users guide and reference manual, version 2.0.0. Technical Report GAP-99-020, Auger project, 1999.

[35] H. J. Song, X. Liu, D. Jakobsen, R. Bhagwan, X. Zhang, K. Taura, and A. Chien. The MicroGrid: a Scientific Tool for Modeling Computational Grids. In *Proceedings of SC2000*, 2000.

[36] J. Stiles, T. Bartol, E. Salpeter, , and M. Salpeter. Monte Carlo simulation of neuromuscular transmitter release using MCell, a general simulator of cellular physiological processes. *Computational Neuroscience*, pages 279–284, 1998.

[37] A. Takefusa, S. Matsuoka, H. Nakada, K. Aida, and U. Nagashima. Overview of a Performance Evaluation System for Global Computing Scheduling Algorithms. In *Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing (HPDC)*, pages 97–104, August 1999.

[38] A. Turgeon, Q. Snell, and M. Clement. Application Placement Using Performance Surfaces. In *Proceedings of the 9th IEEE International Symposium on High Performance Distributed Computing (HPDC)*, 1999.

[39] R. Wolski, N. Spring, and C. Peterson. Implementing a Performance Forecasting System for Metacomputing: The Network Weather service. In *Proceedings of the 1997 ACM/IEEE Supercomputing Conference*, 1997.

[40] T. Zhao and V. Karamcheti. Expressing and Enforcing Distributed Resource Sharing Agreements. In *Proceedings of SC2000*, 2000.

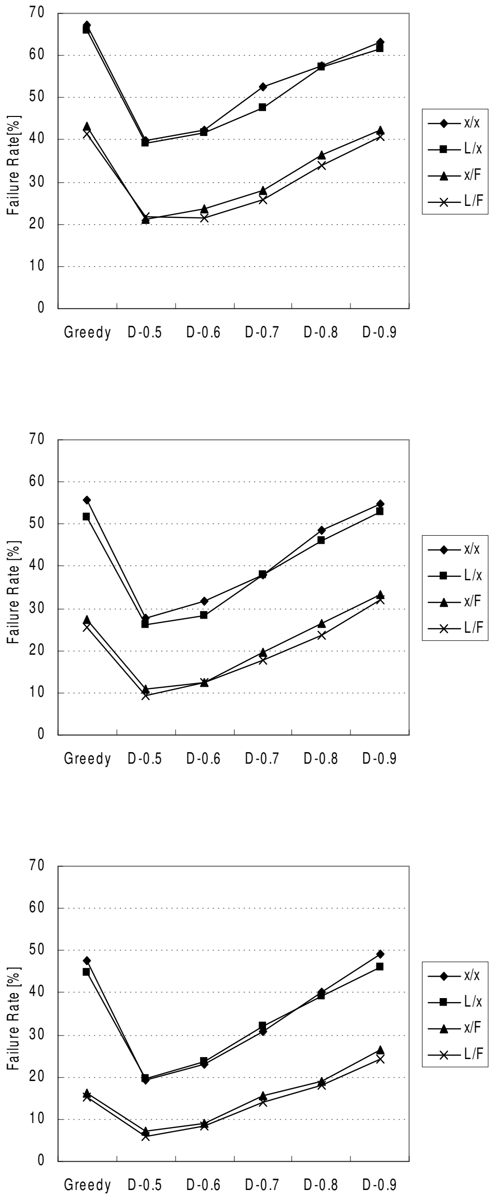

Figure 3. The failure rate for the *Greedy* algorithm and of the *Deadline* algorithm with different values of *Opt* (denoted by D-*Opt* on the x axis) ($N_{max.\ fallbacks}$ = 1, *Int* = 60 (top), 90 (middle), 120(bottom)). Four versions of each scheduling algorithm were used: x/x, L/x, x/F, and L/F where 'L' is Load Correction, 'F' is Fallback, and 'x' is *not used*.

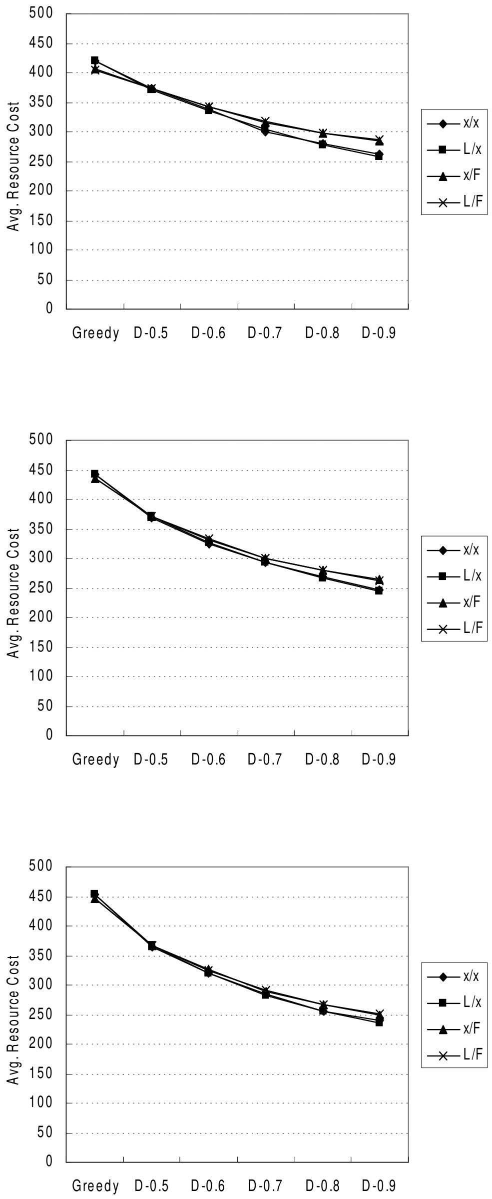

Figure 4. The average resource cost over all requests ($N_{max.\ fallbacks}$ = 1, *Int* = 60 (top), 90 (middle), 120(bottom)).

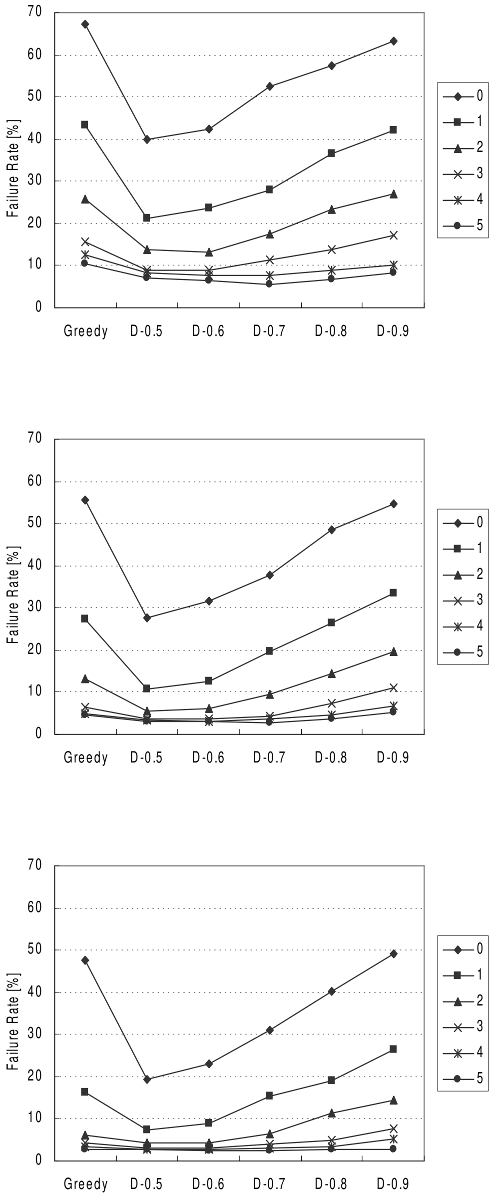

Figure 5. The failure rate for the *Greedy* algorithm and of the *Deadline* algorithm with different values of *Opt* with multiple fallbacks ($N_{max.\ fallbacks}$ = 0, 1, 2, 3, 4, 5, *Int* = 60 (top), 90 (medium), 120(bottom)).

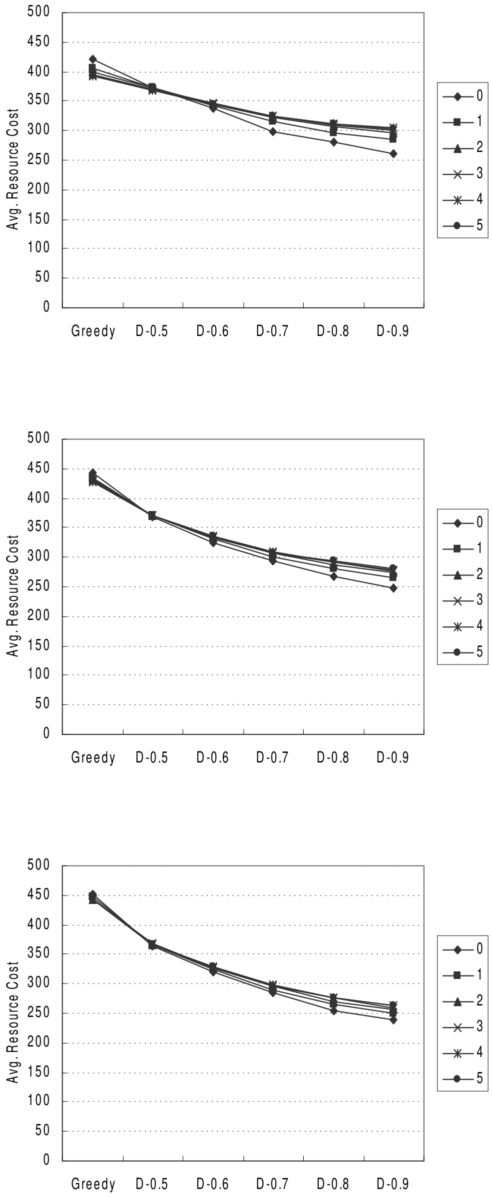

Figure 6. The average resource cost over all requests ($N_{max.\ fallbacks}$ = 0, 1, 2, 3, 4, 5, *Int* = 60 (top), 90 (medium), 120(bottom)).