Quality of Service Scheduling of Computation and Communication Resources in Grid Networks

P. Kokkinos, K. Christodoulopoulos, N. Doulamis, and E. Varvarigos
Department of Computer Engineering and Informatics, University of Patras, Greece and
Research Academic Computer Technology Institute, Patras, Greece.
E-mail: manos@ceid.upatras.gr

Abstract

Grids offer a transparent interface to geographically scattered computation, communication, storage and other resources. In this chapter we propose and evaluate QoS-aware and fair scheduling algorithms for Grid Networks, which are capable of optimally or near-optimally assigning tasks to resources, while taking into consideration the task characteristics and QoS requirements. We categorize Grid tasks according to whether or not they demand hard performance guarantees. Tasks with one or more hard requirements are referred to as Guaranteed Service (GS) tasks, while tasks with no hard requirements are referred to as Best Effort (BE) tasks. For GS tasks, we propose scheduling algorithms that provide deadline or computational power guarantees, or offer fair degradation in the QoS such tasks receive in case of congestion. Regarding BE tasks our objective is to allocate resources in a fair way, where fairness is interpreted in the max-min fair share sense. Though, we mainly address scheduling problems on computation resources, we also look at the joint scheduling of communication and computation resources and propose routing and scheduling algorithms aiming at co-allocating both resource type so as to satisfy their respective QoS requirements.

1 Introduction

1.1 Motivation

Grids consist of geographically distributed and heterogeneous computational, network and storage resources that may belong to different administrative domains, but can be shared among users by establishing a global resource management architecture [1]. A number of applications in science, engineering and commerce, and especially those that exhibit small communication dependencies but large computation and storage needs, can benefit from the use of Grids. An important issue in the performance of Grid Networks is the scheduling of the application tasks to the available resources. The Grid environment is quite dynamic, with resource availability and load varying rapidly with time. In addition to that, application tasks have very different characteristics and requirements. Resource scheduling is considered to be a key to the success of Grids, since it determines the efficiency in the use of the resources and the Quality of Service (QoS) provided to the users.

Why do we need QoS in Grids?

Today's Grids basically provide merely a "best-effort" service to their users. However, this is inadequate if Grids are to be used as the infrastructure for "real world" commercial or demanding applications with strict requirements. Under these thoughts we believe that future Grids will serve two types of users. Some users will be relatively insensitive to the performance they receive from the Grids and will be happy to accept whatever performance they are given. Even though these Best Effort (BE) users do not require performance bounds, it is desirable for the Grid Network to allocate resources to them in a fair way. In addition to BE users, we also expect the Grid Network to serve users that do require a guaranteed QoS. These users will be referred to as Guaranteed Service (GS) users. Grid scheduling algorithms must be able to allocate the resources needed and to coordinate these resources at the right time, right order and in an efficient manner in order to satisfy the QoS requirements of the users and provide fairness among the users. In the content of this chapter we describe and evaluate QoS-aware and fair scheduling algorithms for computation and communication resources in Grid Networks.

1.2 Grid Scheduling Problem

The Grid scheduling problem deals with the coordination and allocation of resources in order to efficiently execute the users' tasks. Tasks are created by applications, belonging to individual users or to Virtual Organizations (VOs), and request the services of the Grid for their execution. Tasks may or may not depend on each other and may require the use of different kinds of resources, such as computation, network, or storage resources, or specific instruments.

The Grid scheduling problem is usually viewed as a hierarchical problem with two levels of hierarchy. At the first level, which is usually called meta-scheduling, a meta-scheduler (also called Resource Broker) selects the resources to be used by a task. These can be computational, communication, storage or other resources. At the second level, which is usually called local scheduling, each resource (more specifically, the local resource management system of the resource) schedules the tasks assigned to it on its local elements. The meta-scheduler and the local scheduler differ in that the latter only manages a single resource, e.g. a single machine, a single network link, a single hard disk. The meta-scheduler receives applications tasks from Grid users and generates task-to-resource schedules, based on various objective functions that it tries to optimize. In this chapter we are more interested in the first level of Grid scheduling, namely meta-scheduling, which is also the most challenging to design and optimize.

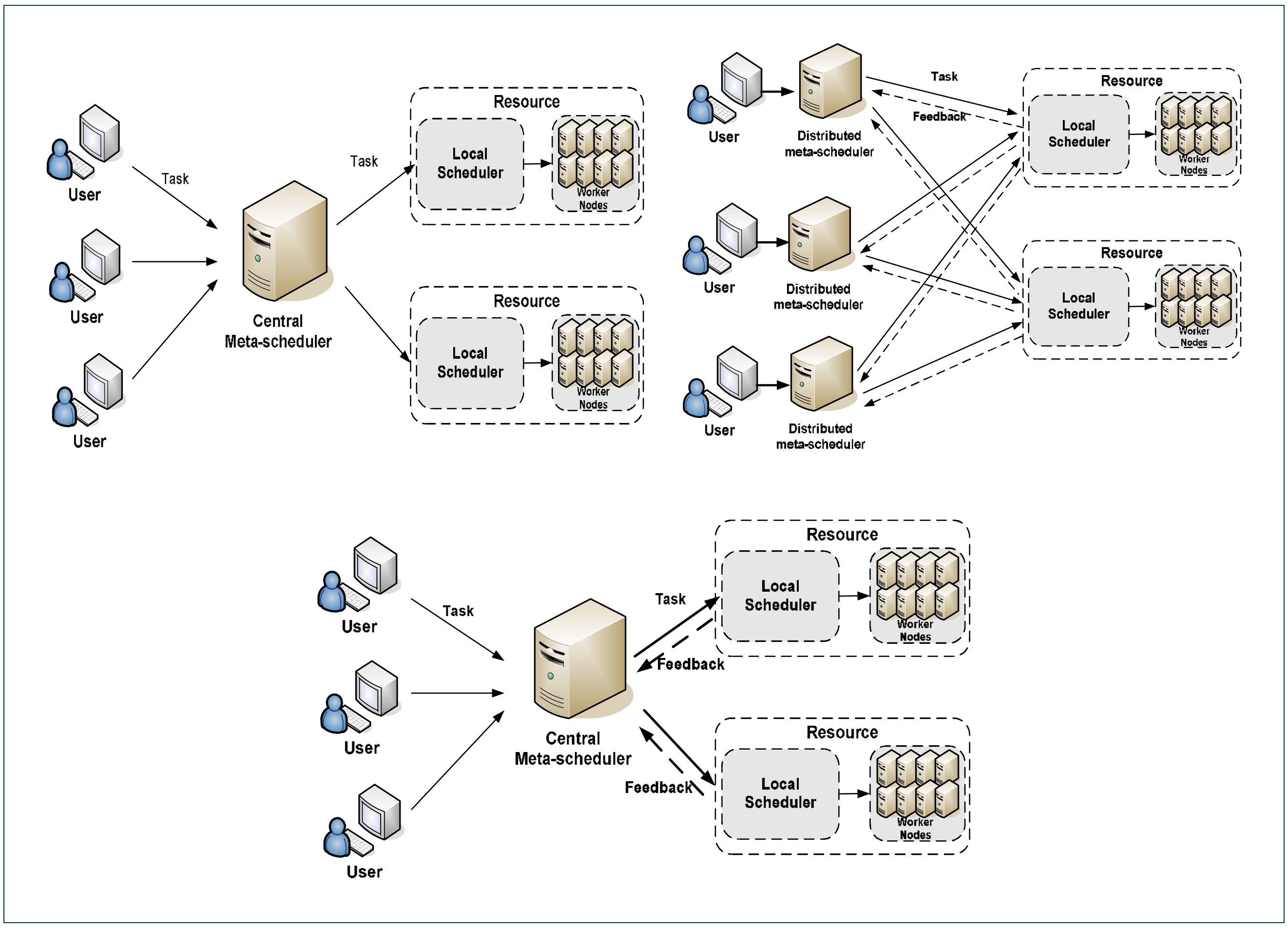

Figure 1 - (a) Centralized (b) Distributed and (c) Hierarchical Grid scheduling architectures

Meta-schedulers can be categorized based on their architecture as *centralized*, *distributed*, or *hierarchical*. In a centralized scheme a central meta-scheduler, located somewhere in the Grid network, collects task requests from all user applications and performs the task-to-resource assignment. In order to assign the tasks efficiently, the central meta-scheduler may use information on the resources utilization and the tasks requirements. In a distributed scheme, there is no central node, but every user site has a meta-scheduler (which we call distributed meta-scheduler) that makes the assignment decision locally. A distributed meta-scheduler communicates with the other distributed meta-schedulers and with the local schedulers in order to exchange resource utilization information so as to improve the efficiency of the employed algorithm. A hierarchical approach considers jointly the meta-scheduling and the local scheduling problems and through the communication of these two levels addresses efficiently the scheduling at both levels. Figure 1 shows block diagrams for the cases of centralized, distributed and hierarchical scheduling architectures.

Meta-schedulers can also be distinguished based on the way they handle new tasks, to *online* and *offline*. Online algorithms assign a task to a resource immediately upon its arrival, while offline algorithms wait for a period of time for several tasks to accumulate at the meta-scheduler, before taking the task-to-resource assignment decisions. The algorithms of the latter type usually consist of two phases: the *task-ordering* phase and the *resource-assignment* phase. In the first phase the order in which the tasks are processed for assignment is determined. In the second phase the resource where a task will be assigned, and possibly the time interval it will use that resource, are selected. Online algorithms can be considered as special cases of offline algorithms where the task-ordering phase uses the First Come First Served (FCFS) queuing discipline. Distributed scheduling schemes usually follow a one phase procedure (online algorithms), while centralized scheduling schemes are employed in one or two phases (online or offline algorithms).

Grid scheduling algorithms handle communication, computation, storage and other kind of resources. A number of algorithms have been proposed that manage these resources either separately or jointly. Although in this chapter we mainly address scheduling problems on computation resources, we also look at joint scheduling of communication and computation resources.

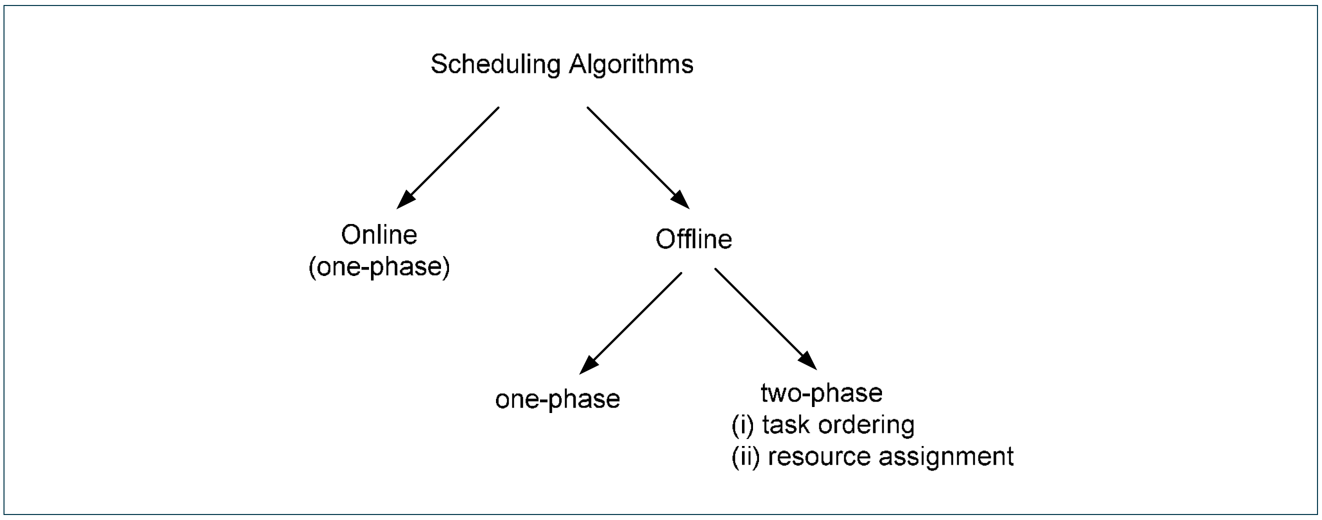

Figure 2 – Categorization of Grid scheduling algorithms into “online” and “offline” algorithms.

1.3 Grid User Requirements and Scheduling Usage Patterns

In order to design efficient and practical Grid scheduling algorithms, we should first consider the QoS requirements that are considered important for the users and the scheduling usage patterns that appear often in a Grid Network.

1.3.1 Grid User QoS Requirements

Grid users have various types of requirements, whether they ask them implicitly or explicitly. Although these include functional requirements, such as security, resource discovery, fault management, performance, event monitoring and others, in this chapter we are mainly interested in non-functional, performance related requirements. The non-functional requirements should be taken into account by scheduling algorithms for selecting a suitable computation site for the execution of a task and a feasible path over which to route the task or the data related to the path, or for performing resources coordination and advance reservations.

Non-functional requirements refer to the Quality of Service (QoS) the user experiences when using the Grid Infrastructure. In general these requirements can be quantitative or qualitative. Qualitative requirements include user satisfaction, service reliability and others. Although qualitative requirements are important, it is difficult to define them and more difficult to measure them objectively. Quantitative requirements on the other hand are easier to define and measure. Some quantitative requirements in which we will be interested in this chapter include:

- **Total Task Delay:** One of the most important Grid user requirements is the total delay of a task, defined as the time that elapses between its creation at a user site and the time its execution results return to the user. Usually, the total task delay consists of two components, the communication and the computation delay. In complicated applications additional delays components exist. Usually the user describes his delay requirements in the form of an upper bound on the total delay of the task he submits. However, separate delay bounds for the computation and the communication delay components may also be requested.

- **Delay Jitter:** Delay jitter is the variation of the total delay of the submitted tasks. In a Grid environment, a Grid user may specify an upper bound on the delay jitter that the underlying computation and communication resources should provide. The smaller the task delay jitter, the better will be the predictability in the use of the Grid and consequently the higher will be the user satisfaction.

- **Bandwidth:** The bandwidth is the rate of data transfer between the Grid user and the computation resource, or between the storage resource (data repository) and the computation resource. The user usually specifies the minimum end-to-end bandwidth it needs for its task and the time period that this bandwidth is requested.

- **Task Rejection (or Blocking) Probability:** The rejection probability is the probability that the task will not be scheduled on the Grid, during the time period the user desires. This can be the result of various network effects, such as contention in the wavelength domain (in WDM networks) or the time domain (packet or burst switched networks). Also if none of the computation resources can serve the task, due to the large number of users already being served the task may also be rejected. Schedulability, defined as one minus the rejection probability, is an alternative performance metric often used.

- **Computational (CPU) Capacity:** The computational (CPU) capacity of a resource is a metric defining how fast the resource can process a task. The computational requirements of a user depend on the way the computation resource is being used – i.e. as a shared (time sharing) or an exclusive access resource (space sharing). In the time sharing approach more than one user-level applications can share a CPU. In this case the user specifies that he requires a certain percentage of the CPU over a particular time period. In the space sharing approach one user-level application has exclusive access to one or more CPUs. In this case the user can specify the number of CPUs and their corresponding speed (measured, e.g., in Million Instructions perm Second, or MIPS). In the space sharing case a user task is allowed to use 100% of the CPU, over a particular time period.

- **Storage Capacity:** The storage capacity is the amount of storage space that is needed by the task's data. Again the user can specify the time period over which the storage space will be needed. Also a user can request a minimum memory size for the task's execution.

- **User's Budget:** The user's budget is the amount of money the user is willing to pay for the execution of a task. This requirement is applicable when the user has to incur a monetary cost for utilizing the resources, in which case the user will not choose a resource combination (network, computation, storage) whose cost exceeds his budget. Indirect payment, e.g. through the allocation of the user's own resources to the Grid, is also possible and probably more widespread at the current time.

- **Application Dependencies:** A user's task may have a number of application dependencies, including the need for a specific operating system, libraries, or other software. These dependencies can be formalized to quantitative requirements by expressing them as boolean variables.

The users may specify one or more of these requirements, while declaring whether they want a best effort or a guaranteed satisfaction of them. Users of the first kind will be referred to as *Best Effort* (BE) users, while users of the second kind will be referred to as *Guaranteed Service* (GS) users. Best Effort users may either not specify any bound on any requirement, or they may specify some, indicating in this way their preferences, but there is no penalty if this requirement is not satisfied. Guaranteed Service users, on the other hand, request a guaranteed bound (upper or lower) on one or more of their qualitative requirements.

Fairness among users (or tasks) is also an inherent requirement for both GS and BE users (or tasks). GS users should receive a graceful and fair degradation in the QoS they experience, if the computation and communication resources are limited. Moreover, it is desirable for the Grid Network to allocate resources to BE users also in a fair way, even though such users do not specify any (other) QoS requirement.

1.3.2 Grid Scheduling Usage Patterns

The Grid Scheduling Architecture Research Group (GSA-RG) of the Open Grid Forum (OGF) in [5] provides different Grid scheduling use case scenarios and describes common usage patterns. These scheduling usage patterns are based on experiences obtained by existing or completed Grid projects. In this section we describe a number of scheduling usage patterns based on the above document and on our own experiences in designing Grid Scheduling algorithms. The contents of the current chapter follow to a great extent the Grid scheduling usage patterns that are presented here. Thus, in each of the following three sections of this chapter we propose algorithms that address the following three usage patterns.

1.3.2.1 *Simple task submission*

This usage pattern corresponds to the case where a user (or an application or a Virtual Organization - VO) submits a simple task on the Grid Network for execution. We assume that this task has no dependencies on data or on other tasks and requires no service guarantees. The user, owner of the task, uses the Grid as a large computational unit where he can submit his task, without the need of utilizing his own possibly less capable resources. The user cares only for the results of the execution of his task.

The Grid scheduler that receives a task with such characteristics is responsible for finding a resource where the task should be executed. There are no dependencies that the scheduler must consider, and no guarantees are required that the scheduler must fulfill. So the scheduler provides a best effort service to the user’s task. Possible criteria the scheduler may use in order to select the most suitable resource for the task’s execution include the resource availability, the current load of the resource, queue lengths or the response time of previous requests, and others. This information can be retrieved by the scheduler by querying either an information service or separately each resource. Finally, after selecting the resource, the scheduler forwards the task to that resource without further processing.

1.3.2.2 *Task requesting a service guarantee*

This usage pattern corresponds to the case where a user (or an application or a Virtual Organization) submits a task that has no dependencies but requires a specific service guarantee. Such service guarantees can be the task’s deadline, the number of CPUs required, the storage size needed to store the execution results, and others. In this usage pattern the user not only uses the Grid Network as an alternative of his own infrastructure but also it uses it because the Grid Network can provide a specific guarantee that his own infrastructure (possibly) cannot. Furthermore, the task has no dependencies and if no resource is found capable of meeting the task requirements then the task is not scheduled.

In order for a resource to be able to provide a service guarantee, a number of mechanisms can be used, including advance (or not) reservations, queuing mechanisms, backfilling strategies, admission control and others. The most often used mechanism is advance reservation, where the local resource manager reserves in advance a resource based on the orders of the Grid scheduler. The resource can be a storage resource, a computation resource, a network resource, a sensor device, an instrument, and others.

The Grid scheduler that receives a task requesting a service guarantee is responsible for finding a resource that can fulfill this request. The information the scheduler needs in order to select a resource can be retrieved by querying either an information service or separately each resource. So we assume that the resources either publish information about the service guarantees they can provide and under which conditions on an information service, or they can reply directly to the Grid scheduler if he queries them. Furthermore, it is possible that a negotiation procedure is used (like e.g. WS-Agreement [28]), between the scheduler and a resource. After a resource is selected and a specific performance guarantee is agreed, the local resource manager is ordered by the Grid scheduler to perform the necessary actions, for example to reserve in advance the resource. Next, the scheduler forwards the task to that resource.

1.3.2.3 *Task requesting many service guarantees*

This usage pattern corresponds to the case where a user (or an application, or a Virtual Organization - VO) submits a task on the Grid Network for execution requesting a number (more than one) of service guarantees. The submitted task can consist of a number of “subtasks”, with various interdependencies. As in the previous case, in this usage pattern the user not only uses the Grid Network as an alternative of his own infrastructure but also it uses it because only the Grid Network can serve his task. If no resources are found capable of serving the task’s requirements then the task is not scheduled.

We can further distinguish this usage pattern in two sub-cases:

(a) **Concurrent**: The guarantees are requested in the same time frame, simultaneously, and we consider concurrent co-allocation of resources. In this usage pattern information is needed about the availability of resources and the respective level of service they can provide in order for the Grid scheduler to plan a synchronized allocation (reservation) of a set of resources. Co-allocation in this case refers to the parallel allocation (reservation) of all resources involved, like e.g. a number of CPUs required for executing an MPI task. Information service and negotiation frameworks are needed.

(b) **Workflow**: The guarantees are requested at different time frames, and we then speak about workflows which generally include different steps to be executed on different resources. In order to be able to handle such (complex) workflows, the Grid scheduler must apply advance reservation of resources with different allocation times. Also, this implies that the Grid scheduler has to take into account the dependencies between the tasks and the different resource requests, and the overall allocation of resources to avoid deadlocks. Information service and negotiation frameworks are needed.

1.4 Our Contribution

In this section we present shortly the Scheduling algorithms proposed in this chapter, and investigate the objectives that they try to satisfy.

In Section 2 we examine a number of fair scheduling algorithms. Fairness can be defined in several ways, but an intuitive notion of fairness is that a user, submitting a task on the Grid Network, is entitled to as much use of the resources (computation, network, storage or other resources) as any other user, provided that he needs and can make good use of his share. For example, the task blocking probability should equally affect all users. Based on this notion of fairness, we believe that fair Grid scheduling algorithms should be used both in cases where a best effort service is required, such as in the “simple task submission” usage pattern described in Section 1.3.2.1, but also in the case where service guarantees are requested as in the usage patterns described in Sections 1.3.2.2 and 1.3.2.3. When the Grid serves multiple classes of users (e.g., users willing to pay different amounts of money), fairness can be dependent on the class of the user, with users belonging to the same class having a fair access to the resources that correspond to that class. The fair scheduling algorithms described in Section 2 are offline and can run either in a centralized or a distributed scheduling architecture. These algorithms follow the two-phase scheduling procedure (task-ordering phase, resource assignment phase) and their target is the fair sharing of the computational capacity of the Grid, while taking into account the task requirements and characteristics.

In Section 3 we propose and analyze a framework for providing hard delay guarantees to Grid users. Delay bounds are the most common requirements users impose on the execution of their tasks. The proposed framework focuses on computational rather than communication resources. In order to provide hard delay guarantees, the users are leaky bucket constrained, so as to follow a constrained task generation pattern, agreed separately with each resource during a registration phase. This way the framework provides deterministic delay guarantees, without actually reserving the computation resources in-advance. Such a framework can be used to serve tasks requesting delay guarantees, and thus fall in the “task requesting a service guarantee” usage pattern, described in Section 1.3.2.2. The proposed framework can be employed in a centralized or a distributed scheduling architecture.

In Section 4 we address the problem of in-advance co-allocating resources in a Grid environment. Thus the algorithms presented in this section falls in the “Task requesting many service guarantees -workflow” usage pattern described in Section 1.3.2.3. We assume that task processing consists of two successive steps: (i) the transfer of data from the scheduler or a data repository site to the computation resource in the form of a timed connection and (ii) the execution of the task at the cluster, defining in this way a simple two-step workflow. We present a multicost algorithm for the joint scheduling of the communication and computation resources needed by such a task. The proposed algorithm selects the computation resource to execute the task and determines the path to route the input data. Furthermore, the algorithm finds the starting times for the data transmission and the task execution and performs advance reservations of the corresponding communication and computation resources. This algorithm can be used for providing network and computation delay guarantees to a task for the simple workflow paradigm defined above. The algorithm presented in Section 4 is an online algorithm that is designed to operate in a distributed scheduling architecture, but can easily be extended to operate in a centralized manner.

2 Fair Scheduling Algorithms for Guaranteed Service and Best Effort Users

Future Grid Networks are expected to serve two types of users. Some users will require Quality of Service (QoS) guarantees, given in terms of an upper bound on the maximum allowable delay (deadline) or on the maximum allowable delay jitter, or a lower bound on the minimum required computational power, etc. Such users will be referred to as Guaranteed Service (GS) users, and the objective of the scheduling algorithms is to provide them with the required QoS level, or if this is not possible due to the limited computation or communication resources, to offer a graceful and fair degradation in the QoS they receive. On the other hand, some users will be relatively insensitive to the performance they receive from the Grid and will be happy to accept whatever performance they are given. Even though these Best Effort (BE) users do not require performance bounds, it is desirable for the Grid to allocate resources to them in a fair way. By the term "user" we do not necessarily mean an individual user, but also a Virtual Organization (VO), or a single application, using the Grid infrastructure.

Most Grid scheduling algorithms proposed to date, schedule tasks based on various task characteristics, such as their deadline, workload, estimated completion time, or price the user is willing to pay for their execution. In [8] the scheduling algorithm proposed tries to minimize the total average task delay and maximize resource utilization. The scheduling algorithms proposed in [2], [13] and [14] try to minimize the total completion time by dropping over-demanding tasks (e.g., tasks of high workload and short deadlines). Other performance metrics used are the average task slowdown [7], defined as the ratio of the task's total delay to its actual run time, the deadline miss ratio [6], and several other metrics. Buyya et all in [3],[4] propose an economic-based approach, where scheduling decisions are made “online” and they are driven by the end-users requirements. In [6] the authors apply economic considerations in resource management and propose GridIS, a Peer-to-Peer (P2P) decentralized scheduling framework. In GridIS a user (consumer) wishing to execute a task sends a task announcement via a portal. The announcement is forwarded throughout the P2P network and the resource providers that receive it bid for the task (auction).

Fairness is an important QoS requirement that should be taken into account in Grid scheduling. The number of different resource types (computation, communication, storage) comprising a Grid makes the enforcement of fairness in such systems a more complex issue than, for example, in Data Networks. In Data Networks the Generalized Processor Sharing (GPS) [15] has been proposed for the fair sharing of capacity on communications links. The GPS scheme provides guarantees on the delay and bandwidth of a session in a network of switches, but is hard to implement. Its approximation, Weighted Fair Queuing (WFQ) [16], is instead often used in Data Networks. WFQ exploits concepts of the Max-Min Fair sharing scheme [17]. GPS-based algorithms are widely implemented in the Internet and mobile communications today. In Grid Networks a number of fair scheduling algorithms have been proposed [18][19].

In this section we present a number of fair scheduling algorithms for Grid Networks. The fair scheduling algorithms to be described are “offline” and consist of two-phases: the task-ordering phase and the resource assignment phase. Offline algorithms wait for a period of time so that several tasks accumulate at the meta-scheduler. When the period expires, tasks are ordered (task-ordering phase) and are assigned to the resources (resource-assignment phase). These algorithms incorporate fairness considerations in one of these two phases, so as to give users fair access to the computational capacity of the Grid. Naturally, the fair sharing of the capacity is influenced by the specific characteristics of the tasks (task lengths, deadlines, user budget). The proposed algorithms run either in a centralized or a distributed scheduling architecture.

In Section 2.1, we propose three new fair scheduling algorithms for Grid computing [22] that take task deadlines into account. These algorithms incorporate fairness in the first of the two phase scheduling procedure, and are based on the Max-Min fair sharing concept. The first, called Simple Fair Task Order (SFTO) algorithm, orders the tasks according to, what we call, their fair completion times. The second, called Adjusted Fair Task Order (AFTO) algorithm, refines the SFTO policy by ordering the tasks using the adjusted fair completion times. Finally, the third scheme we present, called the Max-min Fair Share (MMFS) algorithm, simultaneously addresses the problem of finding a fair task order and assigning a processor to each task based on a Max-Min fair sharing policy.

In Section 2.2 we propose another scheduling algorithm for Grid, called Fair Execution Time Estimation (FETE), which incorporates fairness in the second of the two phase scheduling procedure [55]. Under FETE, tasks are scheduled based on their fair completion time on a certain resource, which is an estimation of the time by which a task will be completed on the resource assuming it gets a fair share of the resource’s computational power. Though space-shared scheduling is used in practice, the estimates of the fair completion times are obtained assuming that a processor sharing discipline is used.

In Section 2.3 we describe a scheduling procedure that provides fairness among users [54] instead of fairness among tasks, as was done in Sections 2.1 and 2.2. In the literature a number of works have supported user fairness in Data [23] or in Grids [20], instead of packet, flow or task fairness. The notion of user fairness is more appropriate for Grids, since the main entities in Grids are not the tasks but the users creating them (a "user" may also refer to a VO). Different weights can also be given to users (or VOs), in which case we talk about weighted user fairness.

2.1 Fair Scheduling Algorithms that take Task Deadlines into Account

In this section we present three fair scheduling algorithms for Grid computing that take the task deadlines into account [22]. These algorithms incorporate fairness considerations in the first of the two phase scheduling procedure, by determining a fair order in which tasks should be considered for scheduling.

2.1.1 Notation and Problem Formulation

We define the workload w_i of task T_i, i=1,2,...,N, as the task duration when executed on a processor of unit computation capacity, where N is the number of tasks to be scheduled. Task workloads are assumed to be known a priori to the scheduler, and may be provided by user estimates or through a prediction mechanism, such as script discovery algorithms, databases containing statistical data on previous runs of similar tasks, or other methods [21]. We assume tasks are non-preemptable, so that when they start execution on a machine they run continuously on that machine until completion. We also assume that time-sharing is not available and a task under service occupies 100% of the processor capacity.

We assume that the computation capacity of processor j is equal to C_j units of capacity and we have a total of M processors. We let d_{ij} be the communication delay between user i and processor j. More precisely, d_{ij} is (an estimate of) the time that elapses between the time a decision is made by the resource manager to assign task T_i to processor j, and the arrival of all files necessary to run task T_i to processor j. Each task T_i is characterized by a deadline D_i, which is the time by which it is *desirable* for the task to complete execution. In our formulation, D_i is not necessarily a hard deadline. In case of congestion, the scheduler may not assign sufficient resources to the task to complete execution before its deadline, in which case the user may choose not to execute the task. We use D_i together with the estimated task workload w_i and the communication delays d_{ij}, to obtain estimates of the computation capacity that task T_i would have to reserve to meet its deadline if assigned to processor *j*. If the deadline constraints of all tasks cannot be met, our target is (i) to obtain a schedule that is feasible with respect to all other constraints, and (ii) the amounts of time by which the tasks miss their respective deadlines to be determined in a fair way.

We let g_j be the estimated completion time of the tasks running on or already scheduled for processor *j*. Therefore, g_j is equal to zero (the present time) when no task has been assigned to processor *j* at the time a task assignment is about to be made and is equal to the remaining time until completion of the tasks assigned to processor *j,* otherwise. We define the *earliest starting time* δ_{ij} of task T_i on processor *j* as

$$d_{ij} = \max\{d_{ij}, g_j\}. \qquad (1)$$

In other words, δ_{ij} is the earliest time at which it is feasible for task T_i to start execution on processor *j*. We define the average of the earliest starting times of task T_i over all the *M* available processors,

$$d_i = \frac{\sum_{j=1}^{M} d_{ij} \cdot C_j}{\sum_{j=1}^{M} C_j}. \qquad (2)$$

We will refer to d_i as the Grid access delay for task T_i, and it can be viewed as the (weighted) mean delay required for task T_i to access the total Grid capacity *C*. Since in a Grid computation power is distributed, d_i plays a role reminiscent of that of the (mean) memory access time in uni-processor computers.

We define the *demanded computation rate* X_i of a task T_i as

$$X_i = \frac{w_i}{D_i - d_i}. \qquad (3)$$

X_i can be viewed as the computation capacity that the Grid should allocate to task T_i for it to finish by its requested deadline D_i if the allocated computation capacity could be accessed at the mean access delay d_i. As we will see later, the computation rate allocated to a task may have to be smaller than its demanded rate X_i. This may happen in case of congestion, when more jobs[1] request service than the Grid can support, and some or all of the jobs may have to miss their deadline. The following fair scheduling algorithms attempt to reduce the computational rates allocated to different tasks in a fair way.

2.1.2 Earliest Deadline First and Earliest Completion Time Rules

The most widely used urgency-based scheduling scheme is the Earliest Deadline First (EDF) method, according to which the system assigns, at any point, the highest priority to the task with the most imminent deadline. The most urgent task is served first, followed by the remaining tasks according to their urgency.

The EDF rule answers the task-ordering question, but does not determine the processor where the selected task is assigned. To answer the resource-assignment question, the Earliest Completion Time (ECT) rule can be used, where the resource *j* that minimizes the completion time $\delta_{ij} + w_{ij}$ of a task T_i is selected.

[1] In this chapter the terms “task” and “job” are used interchangeably.

We have defined γ_j as the processor release time, that is, the time at which all tasks already scheduled on this processor finish their execution. This definition makes it easy to compute γ_j, since it is independent of the task that is about to be scheduled. It has, however, the drawback that gaps in the utilization of a processor are created, resulting in a waste of processor capacity. An obvious way to overcome this problem is to examine the capacity utilization gaps, and see if a task can fit within the corresponding time interval.

2.1.3 Fair Scheduling

The simple scheduling algorithms outlined in section 2.1.2 do not take fairness considerations into account. For example, tasks with relative urgency (with the EDF rule) or tasks that have small workload (with the ECT rule) are favored against the remaining tasks. To overcome these shortcomings, we propose an alternative approach, where the tasks requesting service are queued for scheduling according to what we call their "fair completion times". The fair completion time of a task is found by first estimating its fair task rates using a Max-Min fair sharing algorithm.

2.1.4 Estimation of the Task Fair Rates

Intuitively, in Max-Min fair sharing, all users are given an equal share of the total resources, unless some of them do not need their whole share, in which case their unused share is divided equally among the remaining "bigger" users in a recursive way. In other worlds, tasks demanding small computation rates X_i get all the computation power they require, while tasks demanding larger rates share what is left over. The idea of the Max-Min fair sharing is explained in the example of Table 1, where four tasks with demanded rates 10, 8, 5 and 15 units, respectively, request service, while total offered processor capacity equals 30 units. The total demanded rate of the tasks equals 10+3+5+15=32 units, which is greater than the total offered processor capacity. The max-min fair sharing algorithm aims at reducing the task rates in a fair way so that the assigned task rates equal the total offered processor capacity.

Demanded Rates	Max-Min Fair Sharing (First iteration)	Max-Min Fair Sharing (Second iteration)	Fair Rates
10	7.5	10	10
3	3	3	3
5	5	5	5
15	7.5	11	12
Residue	30-23=7	1	0

Table 1: An example of the non-weighted Max-Min fair Sharing algorithm. Since all tasks are assumed to be of equal importance, the algorithm initially divides the 30 unit total processor capacity into four parts, each of 7.5 units. Tasks 2 and 3 request less than 7.5 units (3 and 5, respectively) and thus get the rate they request. Tasks 1 and 4 demand rate more than 7.5 units (10 and 15, respectively) and are assigned during the 1st iteration a rate of 7.5 units each. Consequently, at the end of the 1st iteration a residue of 30-23=7 units is left to be allocated in the following steps. In the 2nd iteration, the residue of 7 units is equally shared among Tasks 1 and 4, so that each gets an additional 3.5 units. Since Task 1 requires less than 11 (=7.5+3.5) units, it gets the rate it requests, i.e., 10 unit, while Task 4 gets a 11 units at the end of iteration 2 and a residue of 1 unit is obtained, which is given to Task 4 during the next iteration.

It is also possible to assign different priorities to users. More specifically, we assume that each task T_i is characterized by an integer weight j_i, determined, for example, by the user's contribution to the Grid infrastructure, or by the price he is willing to pay for the services he receives. In weighted Max-Min fair sharing scheme the rate a user with weight j_i gets is equal to the total rate that j_i simple users would get in the (non weighted) Max-Min fair sharing scheme.

2.1.5 Fair Task Queue Order Estimation

As mentioned previously, a scheduling algorithm should first choose the order in which the tasks are considered for assignment to a processor (task-ordering), and then, for the task that is located each time at the front of the queue, it should decide the processor on which the task is assigned (processor/resource-assignment). To solve the queueing order problem in fair scheduling, we will describe in Sections 2.1.5.1 and 2.1.5.2, two ordering disciplines of different degrees of implementation complexity.

2.1.5.1 *Simple Fair Task Order (SFTO) scheme*

We define the non-adjusted fair completion time t_i of task T_i as

$$t_i = d_i + \frac{w_i}{r_i}, \tag{4}$$

t_i can be thought of as the time at which the task would be completed if it could obtain constant computation rate equal to its fair computation rate r_i starting at time d_i (recall that d_i is the mean Grid access time for task T_i). Note that finishing all tasks at their fair completion time is unrealistic because the Grid is not really a single computer that can be accessed by user *i* at any desired computation rate r_i at a uniform delay d_i.

According to the Simple Fair Task Order (SFTO) rule, the tasks are placed in the queue in increasing order of their non-adjusted fair completion times t_i, defined in Eq. (4). In other words, the task that is first considered for assignment to a processor is the one for which it would be fair to finish sooner. Note that the non-adjusted fair completion times are obtained from the non-adjusted computational rates r_i, which are in turn estimated from the tasks' demanded rates X_i and the total Grid processor capacity *C*.

2.1.5.2 *Adjusted Fair Task Order (AFTO) scheme*

An issue not addressed in the definition of the non-adjusted fair completion times given in Section 2.1.5.1 is that when tasks complete execution, more capacity becomes available to be shared among the active tasks, and the fair rate of the active tasks should increase. Moreover, when new tasks become active (because of new arrivals), the fair rate of existing tasks should decrease. Therefore, the fair computational rate of a task is not really a constant r_i, as assumed in previous section, but it is a function of time, which increases when tasks complete execution, and decreases when new tasks arrive. By accounting for this time-dependent nature of the fair computational rates, the adjusted fair completion times, denoted by t_i^a can be calculated, which better approximates the notion of max-min fairness. In the Adjusted Fair Task Order (AFTO) scheme, the tasks are ordered in the queue in increasing order of their adjusted fair completion times t_i^a. The AFTO scheme results in schedules that are fairer than those produced by the SFTO rule; it is, however, more difficult to implement and more computationally demanding than the SFTO scheme, since the adjusted fair completion times t_i^a are more difficult to obtain than the non-adjusted fair completion times t_i. An algorithm to compute the adjusted fair completion times t_i^a is given in [22].

2.1.5.3 *Processor Assignment*

The SFTO scheme or the AFTO scheme is used to determine the order in which the tasks are considered for assignment to processors, but it still remains to determine the particular processor where each task is assigned. A simple and efficient way to do the processor assignment is to use the earliest completion time rule (ECT), modified so that it exploits the capacity gaps (Section 2.1.2).

2.1.6 Max-Min Fair Scheduling (MMFS) Scheme

In this section, we present an alternative fair scheduling scheme that simultaneously obtains a fair task queuing order and a fair processor assignment. In this algorithm, our goal is to assign a schedulable (actual) rate r_i^S to each task so that it is as close as possible to its fair task rate r_i (derived by applying the max-min fair sharing algorithm on the demanded rates X_i, as described in Section 2.1.4). The schedulable rates r_i^S are smaller or equal to the task fair rates ($r_i^s \le r_i$) and they are chosen so as not to violate the processor capacity constraints. This is expressed in the following constrained optimization problem

$$\min E = \min \sum_{i=1}^{N} \left| r_i^S - r_i \right| \tag{5a}$$

subject to

$$\sum_{i \in P_j} r_i^s \le C_j \qquad P_j = \{i : T_i \text{ scheduled on } j \text{ processor}\} \tag{5b}$$

The set P_j contains all tasks scheduled on processor *j*.

The minimization of Eq. (5a) subject to the constraint of Eq. (5b) can be performed using the algorithm described in [22]. The main idea is to perform an initial processor assignment and then, appropriately re-arrange tasks on underused and overused processors so as to better exploit the processor capacity.

2.1.7 Performance Results

In this subsection we compare the performance of the scheduling algorithms presented in Section 2.1 to that of other traditional scheduling policies. More detailed simulation experiments are presented in [22].

We define the normalized load of the Grid infrastructure as the ratio of the tasks' demanded computational rates X_i over the total processor capacity *C* offered by the Grid infrastructure:

$$r = \frac{\sum_{i=1}^{N} X_i}{C} \, . \tag{6}$$

A Grid with load r is able to serve, on average, *N* tasks of workload w_i, deadlines D_i and ready times d_i within a time interval of $\Xi = r * E\{D_i - d_i\}$, where the $E\{\cdot\}$ denotes the expected value. In the arrival model adopted, we assume that the *N* tasks arrive at the Grid into *β* groups, each of N / b tasks. We also assume that the arrival times of each group of N / b tasks follows the Poisson distribution with parameter *λ*, where $l^{-1} = b \cdot \Xi$. In our experiments *β*=10.

The criterion we used for measuring scheduling performance is the deadline fairness metric, which is the average relative deviation of the demanded task deadlines to the actual task completion times, defined as:

$$E = \frac{1}{N} \sum_i \frac{\left| D_i - \max(D_i^c, D_i) \right|}{D_i} \, , \tag{7}$$

where D_i is the requested deadline and D_i^c is the actual completion time of the *i*th task. Tasks whose actual completion times are smaller than their respective deadlines do not contribute to *E*.

The FCFS and EDF algorithms do not permit any violations of the task deadlines and they may reject tasks, in which case the error *E* becomes equal to infinity. To overcome this difficulty, we assume that tasks whose deadlines are violated are put in a waiting list, and reapply for execution.

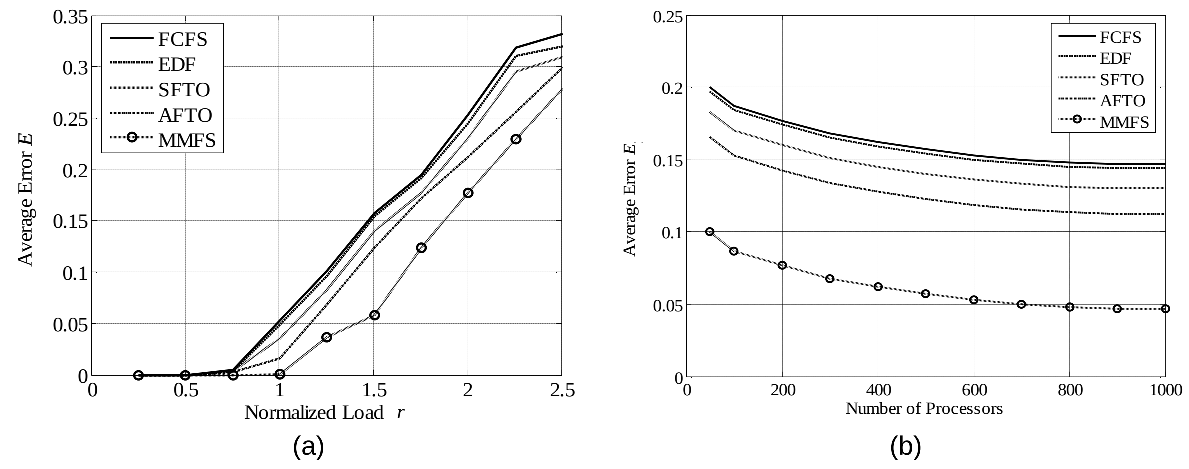

Figure 3 – (a) The error *E* versus the normalized load *p* for the FCFS, the EDF, the SFTO, the AFTO and the MMFS algorithms, and (b) the error *E* versus the number of processors for *ρ*=1.5 and workload variance 0.1.

Figure 3.a illustrates the error *E* obtained for the SFTO, AFTO, and MMFS scheduling policies against the normalized load r . For comparison purposes, we also depict in Figure 3 the results obtained for the FCFS and EDF schemes. The simulations assumed a Grid consisting of 500 processors of almost equal capacity (*symmetric processor case)*. In particular, we assume that the capacity of all the 500 processors follows a normal distribution with standard deviation of 1% of the respective mean capacity value. We assume that 2500 tasks arrive at the Grid within a time interval of duration Ξ, and the experiment is repeated for 20 time intervals Ξ, for a total of 2500*20=50000 tasks. In this experiment, we assume that tasks have almost similar deadlines with a normal distribution of 1% standard deviation of the respective mean deadline value. The mean value of the task workloads (task sizes) varies, so that the normalized load r takes values from 0.25 to 2.5, while the respective standard deviation equals 10% the mean value. We observe that the MMFS scheduling policy yields the highest efficiency while the AFTO algorithm is the second best. The worst performance is obtained by the FCFS policy. For light load ($r < 0.6$), all algorithms efficiently schedule the tasks, but as the load r increases, the MMFS policy outperforms the other schemes.

Figure 3.b illustrates the performance of the five examined scheduling policies when the number of processors varies between 5 and 1000. In all cases, symmetric processors are assumed, and the load is r =1.5. The workload variance is assumed to equal 10% of its respective mean value. As the number of processors increases, the number of tasks is increased to keep the load constant. We see that the MMFS policy outperforms the other algorithms, with the AFTO scheme giving the second best performance. As the number of processor increases a slight improvement in scheduling efficiency is observed.

2.2 Fair Execution Time Estimation (FETE)

In this section we propose an alternative fair scheduling algorithm for Computational Grids, which we call the Fair Execution Time Estimation (FETE) algorithm [55]. FETE assigns a task to the resource that minimizes what we call its fair execution time estimation. This estimation is obtained assuming that the task gets a fair share of the resource's computational power. Though space-shared scheduling is used in the actual system, the estimates of the fair execution times are found assuming time-sharing (processor sharing) is used. FETE approaches to the closest extent the notion of weighted max-min fairness. Next we consider the possibility of incorporating FETE in a production Grid Middleware. For this reason we propose a related scheduling scheme, called "simple FETE", which is a good approximation of FETE, and does not require a-priori knowledge (or estimates) of the task workloads. FETE and simple FETE can be implemented both in a centralized and in distributed way.

2.2.1 Grid Network Model

We consider a Grid Network that consists of a number of users and a number of resources. For the sake of being specific, we assume a centralized implementation of the FETE algorithm, so a single central scheduler exists. We assume that the communication delays are negligible compared to the task execution times. Users generate undivisible and non-preemptable tasks with varying characteristics. Every task *i* has workload w_i and a non-critical deadline D_i. By "non-critical" we mean that if the deadline expires, the task remains in the system until completion, but it is recorded as a deadline miss. Each resource *j* contains a number CPUs, of total computational capacity equal to C_j, that use a space-sharing policy. Each resource has also a local queue and a local scheduler. Tasks arriving at the resource are stored in its queue, until they are assigned by the local scheduler to the available CPUs. We assume that the local scheduler uses the First Come First Serve (FCFS) policy, where tasks are processed in the order they arrive at the scheduler Also, at any time *t* there are $N_j(t)$ tasks in resource's *j* queue or under execution in its CPUs.

2.2.2 Fair Execution Time Estimation Scheduling Algorithm

The Fair Execution Time Estimation (FETE) scheduling algorithm assigns task *i* to resource *j* that provides the minimum fair execution time X_{ij}. The fair execution time X_{ij} is an estimation of the time required for task *i* to be executed on resource *j*, assuming it gets a fair share of the resource's computational power. The estimates X_{ij} are obtained assuming a time-sharing discipline, though space-shared scheduling is used in the actual system. By fair share we mean that at each time *t* the task gets a portion $1/(N_j(t)+1)$ of resource's *j* computational capacity C_j, as if ideal time-sharing (processor sharing) was used. The parameter $N_j(t)$ is the total number of tasks already assigned to resource *j* at the time *t* the assignment decision is made. Obviously, $N_j(t)$ changes with time, increasing by 1 every time a new task is assigned to resource *j* and decreasing by 1 each time a task completes service at resource *j*. As a result, the fair share of the resource's capacity each task gets also changes with time. So, during the estimation of the fair execution time of a task, the estimated completions of the tasks already assigned to the resource should be taken into account. However, in this calculation it is not possible to also incorporate future task arrivals to the resource, which would change the fair share of existing tasks. The fair execution time estimations of the tasks are calculated only once and are not re-estimated when new tasks arrive.

2.2.3 Simple Fair Execution Time Estimation Scheduling Algorithm

The FETE algorithm of Section 2.2.2 is not easy to implement, since it assumes a-priori knowledge of the task workloads. In this section we propose a simple version of FETE, called simple FETE. The simple FETE assigns task *i* to the resource *j* that provides the minimum simple fair execution time $\hat{X}_{ij}$, defined as

$$\hat{X}_{ij} = w_i \cdot \frac{(N_j + 1)}{C_j},$$

where w_i is the workload of task *i*, C_j is the computational capacity of resource *j*, and N_j is the number of tasks in the resource's queue, including the one being processed at the time. The simple fair execution time $\hat{X}_{ij}$ is an estimation of the time that will be required by task *i* to be executed on resource *j*, assuming it gets a fair share of the resource's computational power, without taking into account the possible completions of the other tasks assigned to that resource. Since w_i does not affect the minimization over the resource *j,* the a-priori knowledge of the task workloads is not needed for the simple FETE.

2.2.4 Performance results

The proposed FETE and simple FETE scheduling algorithms, along with other well-known algorithms were evaluated using the GridSim [25] simulator. In our experiments, we assumed a centralized and "offline" implementation of the scheduling algorithms. We also assumed that FETE and simple FETE use the FCFS ordering policy, where tasks are processed (assigned to resources) in the order they arrive.

FETE and Simple FETE were compared to the well-known algorithms presented in Table 2. In the Earliest Deadline First (EDF) ordering policy the task with the most imminent deadline is scheduled first, while in the Least Length First (LLF) ordering policy, the task with the smallest workload is given priority. The Earliest Completion Time (ECT) policy, assigns a task to the resource where the task will finish its execution earlier.

Algorithm	Ordering Policy	Assignment Policy
FCFS/ECT	First Come First Served (FCFS)	Earliest Completion Time (ECT)
EDF/ECT	Earliest Deadline First (EDF)	Earliest Completion Time (ECT)
LLF/ECT	Least Length First (LLF)	Earliest Completion Time (ECT)

Table 2. The scheduling algorithms compared with the FETE and the simple FETE algorithms.

Each user creates 2000 tasks with non-critical deadlines, and all tasks remain in the system until completion even if their deadline expires. The task characteristics are defined probabilistically as shown in Table 3. In our simulations, we evaluated the proposed algorithms by altering, in each experiment, the task submission rate that is the load the users induce on the Grid Network. However, in each experiment all the users have the same task submission rates.

Characteristic	Distribution	Mean or Min/Max
Task Workload	Exponential	14000 MIPS
Task File Size Input	Uniform	1000 bytes / 10000 bytes
Task File Size Output	Uniform	1000 bytes / 10000 bytes
Task Submission Rare	Exponential	12, 20, 25, 33, 40, 50, 55, 60, 65, 70 tasks/sec
Task Deadline	Uniform	10 sec / 20 sec

Table 3: Tasks characteristics.

To assess the performance of the algorithms we used the following metrics:

- The average delay of the tasks (task Delay = task Finish Time - task Creation Time).
- The standard deviation of the task delays.
- The average excess time. Defined as the average time by which a task misses its non-critical deadline (task Excess Time = task Finish Time - task Deadline Expiration).
- The excess time standard deviation.
- The number of non-critical deadlines missed.
- The deadline fairness metric, as defined in Eq. (7).

For all scheduling algorithms examined, the average task delay increases when the task submission rate increases. For light load, below 40 tasks/sec, all scheduling algorithms result in similar average task delay. However, when the task submission rate increases beyond 40 tasks/sec the FETE and the simple FETE algorithms achieve smaller average task delay than the other algorithms considered (Table 2). We also observe that the FETE and the simple FETE algorithms result it smaller task delay variance than the other scheduling algorithms, an indication that the proposed FETE algorithms treat tasks in a more fair manner.

From Figure 4.a we see that the average excess time increases as the task submission rate increases. However, the increase is smaller when the FETE and the simple FETE algorithms are used, meaning that the times by which the tasks miss their deadlines are smaller than when the other algorithms are used. Figure 4.b shows that the excess time standard deviation for the FETE algorithms is very small compared to that of the other algorithms, which means that the tasks that miss their deadline are treated rather fairly. When the task submission rate increases, the number of tasks that miss their deadlines also increases, for all the scheduling algorithms examined, but fewer tasks miss their deadlines when they are scheduled using the FETE algorithms than when they are scheduled with other algorithms. The difference is more pronounced for large task submission rates. Similar results are observed for the deadline fairness metric.

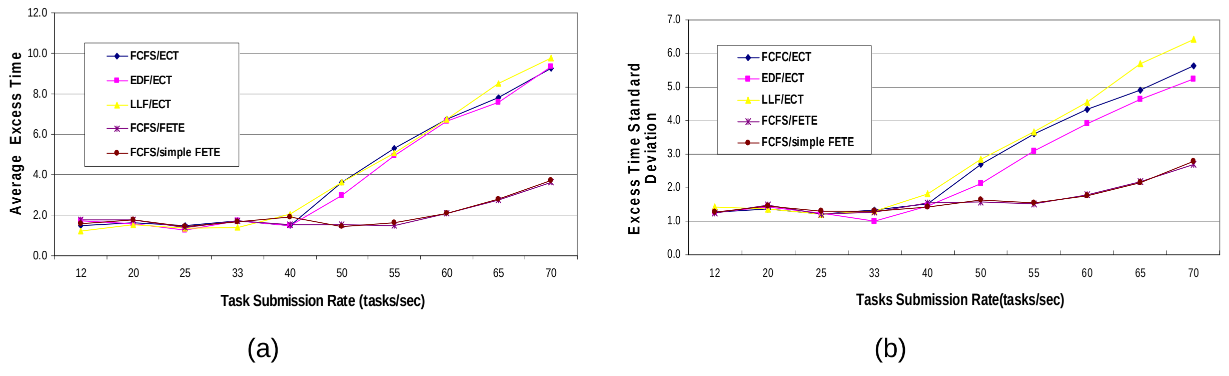

Figure 4 – (a) The average excess time and (b) the excess time standard deviation versus the task submission rate.

2.3 User Fair Scheduling Algorithm

In this section we present a user fair scheduling algorithm for Grids [54]. The fair scheduling algorithms presented in Sections 2.1 and 2.2 provide fairness on a *per task* basis, by sharing in a fair way the computation capacity of the Grid among the various tasks. However, the main entities in Grids are not the tasks but the users (or VO) creating them, so the notion of *user* fairness as opposed to *task* fairness seems more appropriate for Grids. For example, it is not fair for a task belonging to a user who creates only this task, to be handled equally with the possibly thousands of tasks created by some other user. Our proposed scheme provides fairness on a per user basis. It draws on corresponding schemes and concepts developed for Data Networks and specifically on the Weighted Fair Queuing (WFQ) scheduling algorithm [16].

2.3.1 WFQ/EST Scheduling Algorithm

We propose a centralized *user fair* scheduling algorithm for Grids, called WFQ/EST. The WFQ/EST algorithm consists of two phases (task-ordering and resource-assignment), and is executed at periodic intervals (Figure 5). User fairness is mainly achieved in the first, “task-ordering” phase, while in the second any “resource-assignment” algorithm can be used. During a period, tasks belonging to different users arrive at the central meta-scheduler and are handled by a Weighted Fair Queuing (WFQ) scheduler [16]. The WFQ scheduler places these tasks in different queues based on their originating users. When a period expires, the task-ordering phase is executed, during which the tasks are dequeued from the WFQ scheduler and proceed to the “resource-assignment” phase. In the resource-assignment phase the Earliest Starting Time (EST) algorithm is used to assign tasks to resources, however any other algorithm can also be used.

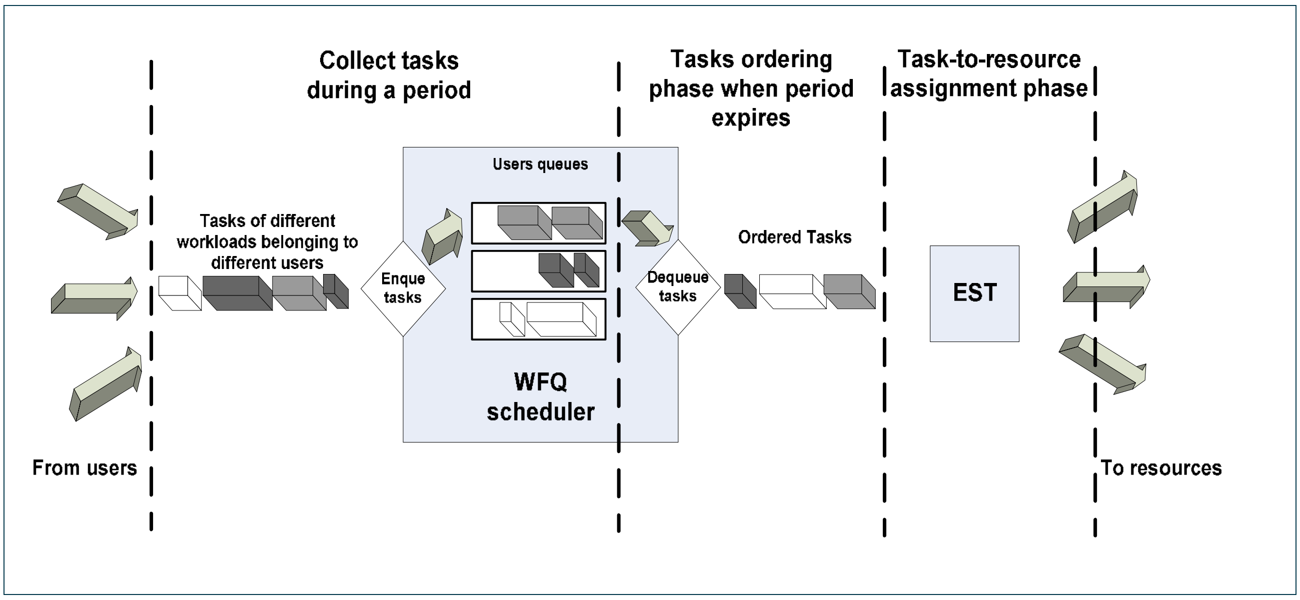
Collect tasks during a period
Tasks ordering phase when period expires
Task-to-resource assignment phase
Users queues
Tasks of different workloads belonging to different users
Enque tasks
Dequeue tasks
Ordered Tasks
EST
WFQ scheduler
From users
To resources

Figure 5 – User fair scheduling using a WFQ scheduler.

2.3.2 Performance Results

We conducted a number of simulations in order to validate that the proposed algorithm provides fairness to the users. In our simulations we used 5 users creating tasks with the characteristics shown in Table 4, 3 single-CPU resources (R1, R2, R3 with total computational capacity 780.000MIPS, 520.000 MIPS, and 260.000MIPS, respectively), and a central meta-scheduler. The resources use a space-sharing policy. We also assume that the resources, the users and the central meta-scheduler communicate directly with each other over links of equal bandwidth and zero propagation delay. All users have non-critical deadlines equal to 110 sec. If a non-critical deadline expires the task remains in the Grid, but is recorded as a deadline miss. Finally, the sizes of the data sent to a resource before task execution and the sizes of the data produced by a resource when the task is completed are the same for all users and equal to 1000 bytes.

User	Task Workload	Task Inter-arrival
U1	Fixed: 419900000 MI	Fixed: 546 secs/task (s/t)
U2	Fixed: 71383000 MI	Fixed: 3278 secs/task (s/t)
U3	Fixed: 419900 MI	Fixed: 6010 secs/task (s/t)
U4	Fixed: 209950000 MI	Fixed: 5464, 4371, 3278, 2186, 1366, 1093, 820, 546, 55 secs/task (s/t)
U5	Fixed: 2099500000 MI	Fixed: 5464 secs/task (s/t)

Table 4: Users task workload and inter-arrival time.

The meta-scheduler's two-phase procedure is executed every 1 sec. In the experiments conducted we compared the WFQ/EST with the EDF/EST scheduling algorithm, using both fairness and performance metrics. The EDF/EST is also a two-phase scheduling algorithm, which uses Earliest Deadline First (EDF) algorithm for the first phase and Earliest Starting Time (EST) algorithm for the second phase.

To asses the performance of the proposed algorithm we used the following metrics:

- The average delay of the tasks. Defined as the average delay of a computation unit (taken to be 1 MI) of a task's workload. A computation unit's delay is defined as the time between its creation as part of a task, and the time this task's execution results return to the user.

- The standard deviation of the task delays. Defined as the standard deviation of the computation unit delay.
- The deadline fairness metric, as defined in Eq. (7).

Table 5 shows the average delay and the standard deviation of a computation unit (taken to be 1 Million Instructions, or MI), for task inter-arrival time of BE user U4 equal to 3278 secs. The average delay of a computation unit metric is more appropriate than the average task delay for comparing the service received by each BE user, given that they generate tasks of different workloads. We observe that the WFQ/EST algorithm achieves smaller average delay of a computation unit than the EDF/EST algorithm. Also, we observe that the standard deviation of the computation unit delay is smaller for the WFQ/EST fair scheduling algorithm than for the EDF/EST scheduling algorithm, indicating that the former algorithm allocates computational capacity more fairly among the users. Furthermore, in Figure 6 we observe that the WFQ/EST also outperforms the EDF/EST algorithm in terms of the deadline fairness metric. The difference with respect to this metric increases as the BE user's U4 task inter-arrival times decrease, and more BE tasks miss their deadlines. So when more tasks are produced than the Grid can serve, the WFQ/EST scheduling algorithm gives a more fair degradation of the probability to miss a deadline among BE tasks.

	EDF/EST	**WFQ/EST**
Average delay (sec)	21.96	13.63
Standard deviation (sec)	48.48	30.07

Table 5: The average delay and the standard deviation of a computation unit using the BE resource scenario, for task inter-arrival time of BE user U4 equal to 3278 sec.

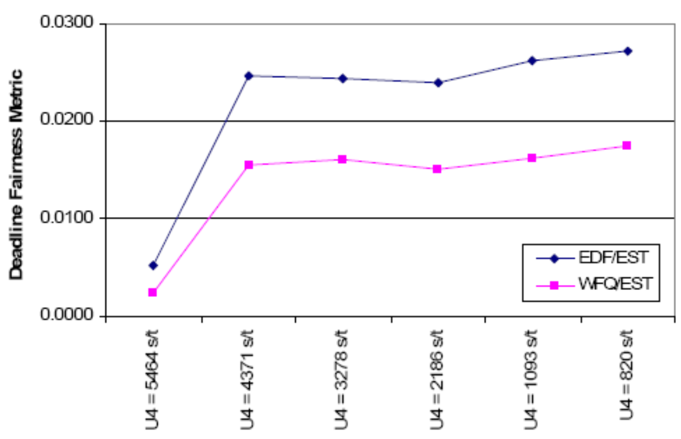

Figure 6 – The deadline fairness metric, for various task inter-arrival times (sec/task) of BE user U4.

3 Framework for Providing QoS Guarantees using Traffic Constrains

In Grid Networks, QoS mainly refers to the total time it takes for a user task to be completed. It can also refer to the time period over which a number of computational resources (space-sharing), or a percentage of one resource (time-sharing), are reserved by a user. Generally, in order for a network to provide QoS guarantees to a user, a three step procedure is followed. At first the user informs the network of the QoS parameters requested (delay, required resources, etc). Then the network, through a procedure called admission control, checks if it can satisfy the user's request for guaranteed service, without violating the guarantees given previously to other users. If this is possible, various mechanisms (resource reservation, scheduling, flow control) are employed to ensure that the agreed upon QoS level is provided to the user.

QoS in Data Networks has been extensively studied. The Internet Engineering Task Force (IETF) had proposed the Integrated Services (IntServ) [9] and the Differentiated Services (DiffServ) architectures [10]. Both architectures support QoS and provide guarantees in terms of bandwidth, latency and other data transfer parameters. Relatively recently there has been an increasing interest in QoS in Grids. Until now two main efforts addressing this issue were presented. The first is the General-purpose Architecture for Reservation and Allocation (GARA) [11] and Grid QoS Management (G-QoSM) [14]. These works propose QoS frameworks for the Grid Networks considering the network, the computational and the storage resources and reserve for a time period resources quantitatively (e.g. reserve a number of CPUs in a resource or reserve a percentage of a CPU's capacity through the Dynamic Soft Real-time scheduler - DSRT [13]). Various other works have concentrated on specific aspects of QoS in Grids, such as network QoS for Grid applications [12], admission control [14] and other.

In this section we propose a QoS framework for Grid computing [52],[53] and [54]. We call it a framework because it gives conditions that if satisfied can provide delay guarantees to Guaranteed Service (GS) users for the completion of a task on a given resource, but leaves a great deal of flexibility in terms of the specific scheduling algorithms to be used. A task's delay is defined as the time between the task's creation and the time its execution results return back to the user. The delay guarantees imply that a GS user can choose a resource to execute his task before its deadline expires, with absolute certainty. The framework can also provide fairness to Best Effort (BE) users, using the user fair scheduling algorithm presented in Section 2.3.

We show theoretically and experimentally that hard QoS, in terms of delay bound guarantees given to each user, can in fact be provided without hard resource reservations. Instead, the GS users are leaky bucket constrained, so as to follow a constrained task generation pattern, which is agreed separately with each resource during a registration phase. This way a user and a resource simply agree upon the task load the former will generate and the latter will serve. In contrast, the GARA and G-QoSM frameworks reserve computational resources explicitly (DSRT [13]). Furthermore, in our QoS framework various other useful features are investigated. We consider single and multi-CPU resources, and scheduling without a-priori knowledge of task workload. We also propose and evaluate computational resources that serve GS, or BE, or both types of users, with varying priorities. Finally, in our simulations data from a real Grid Network are used, validating in this way the appropriateness of the proposed framework.

3.1 Description of the Framework

We consider a Grid Network consisting of a number of users and resources. There are two kinds of users: Guaranteed Service (GS) and Best Effort (BE) users, who generate tasks of GS or BE type, respectively. Also there are various types of resources based on the types of tasks they serve (GS or BE or both) and on the priority they give to each type. In order to service guarantees to GS users, the GS users are leaky bucket constrained, so as to follow a constrained task generation pattern that is agreed separately with each resource. On the resources, the arriving tasks are queued in a Weighted Fair Queuing (WFQ) scheduler [16]. This way guaranteed task service rates (e.g. measured in Millions Instructions Per Second) and guaranteed task delays can be given to each GS user, in the same way WFQ provides guaranteed bandwidth and packet delay services in Data Networks. BE users, on the other hand, are handled by our framework with fairness as the main goal. Algorithms presented in Section 2 can be incorporated for this reason in the proposed framework.

In the following subsections we describe the distributed mechanisms used to provide service guarantees to GS users. We assume, unless otherwise stated, that a task executing at a resource is non-divisible and non-interruptible (non-preemptable). We initially describe our framework assuming that each machine has a single CPU, and later extend it to the multi-CPU machine case.

3.1.1 Guaranteed Service (GS) Users

In the proposed QoS framework, a GS user must first register to a resource, before it can actually use it. During the registration phase, the GS user (or Virtual Organization, VO) and the resource agree upon the characteristics of the computational workload the GS user will send to that resource, that is, the leaky bucket's parameters. A GS user can register to a number of resources. When a GS user creates a task, one of his registered resources is chosen for its execution, based on various criteria, such as performance (e.g., delay), fairness among resources (e.g., uniform utilization of the registered resources), etc.

Our framework is implemented in a distributed way, and, as a result, scheduling logic exists at the GS user site and at the resource site (local scheduler). During the registration phase, a GS user *i* and a resource *r* agree upon the (r_{ir}, s_{ir}) constraints (Figure 7) of the user. The parameter r_{ir} is the long term workload generation rate, measured in computation units per second (e.g., Million Instructions Per Second - MIPS), that GS user *i* will submit to resource *r*. The parameter s_{ir} is the maximum size of tasks (burstiness) that GS user *i* will ever send, in a very short time interval, to resource *r*, and is measured in computation units (e.g., Million Instructions - MI). If resource *r* can accept this average load and burstiness, the GS user is registered to the resource. Thereafter, the GS user becomes responsible for the observance of these constraints and the resource for the satisfaction of the QoS guarantees given to the user, as explained below. Alternatively, other approaches can be used (such as the centralized and the hybrid approaches described in Section 3.2.1), where a meta-scheduler is used as an intermediary for the monitoring of the observation of the *(ρ, σ)* constraints.

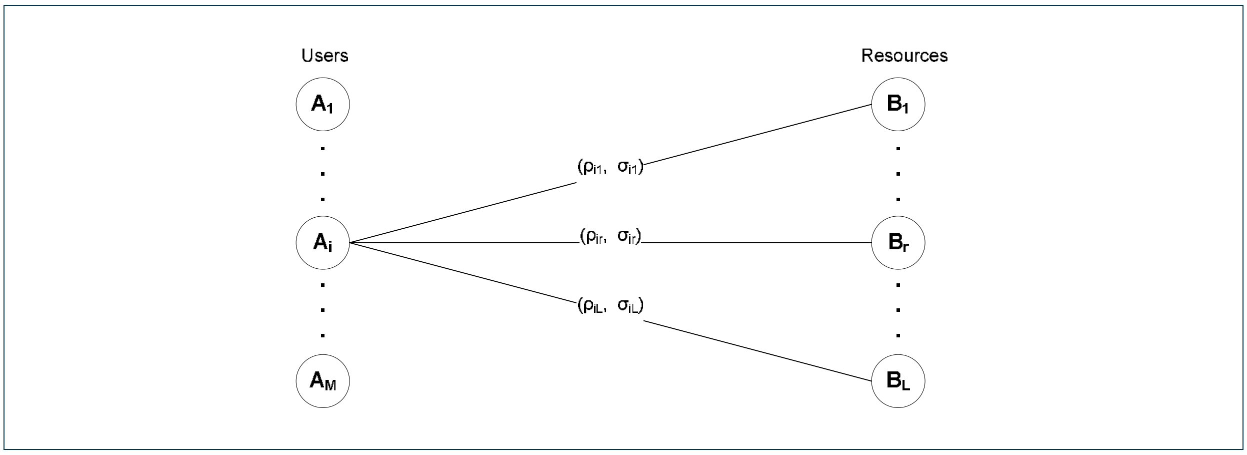
Users
Resources
A_1
A_i
A_M
B_1
B_r
B_L
(ρ_{i1}, σ_{i1})
(ρ_{ir}, σ_{ir})
(ρ_{iL}, σ_{iL})

Figure 7 – The (*ρ*, *σ*) constrained GS users in the Grid Network.

In order for a resource *r* to accept the registration of GS user *ı*, a number of requirements must be met. First, the resource checks whether it can serve the GS user with the requested computational workload generation rate r_{ir} without violating the workload generation rates agreed with the already registered GS users. The local scheduler of every resource applies Weighted Fair Queuing (WFQ) to the queued tasks, so the following condition must hold for new and old GS users:

$$r_{ir} \le g_{ir}(t) = \frac{C_r \cdot w_{ir}}{\sum_{k=1}^{N_r(t)+1} w_{kr}}, \qquad (8)$$

where C_r is the computing capacity of resource *r*, $N_r(t)$ is the number of GS users registered to resource *r* at time *t*, and w_{ir} is the weight of user *i* in using resource *r*. The weights w_{ir} can depend on various parameters, such as the prices the users are willing to pay, or their other contributions to the Grid, Eq. (8) ensures that resource *r* can satisfy the task generation rates of both new and old GS users.

An other condition agreed upon the registration of GS user *i* to resource *r* is that the maximum task workload $J_{ir}^{\max}$ sent by *i* to resource *r* will never exceed the resource's maximum acceptable task workload:

$$J_{ir}^{\max} \leq J_{r}^{\max} . \tag{9}$$

If both Eq. (8) and Eq. (9) hold then the GS user can register to the resource; otherwise, the registration fails and the user must search for another resource. The GS user can repeat this procedure so as to register to multiple resources. Also a user can cancel its registration whenever he wants and for whatever reason. Finally, every user can repeat periodically the registration phase, in order to register to new resources or to resources from which other users have canceled their registrations.

Each GS user *i* is equipped with an input queue to temporarily withhold tasks that if submitted to a resource *r* would invalidate the agreed (r_{ir}, s_{ir}) constraints. Specifically, we denote by $J_{ir}(t)$*, i = 1,2, … ,N* the total computational workload (measured, e.g., in MI) submitted by GS user *i* to resource *r* in the interval *[0, t]*. We will say that GS user *i* is (r_{ir}, s_{ir}) controlled with respect to resource *r*, if the following condition is valid:

$$J_{ir}(t) < s_{ir} + r_{ir} \cdot t, \ \ \forall t > 0 . \tag{10}$$

If a GS task *j* invalidates Eq. (10), the GS user must locally withhold the task for a time period T_{ir}^{j} , until Eq. (10) becomes valid again (Figure 8). So our framework includes in every GS user an admission control mechanism, to make sure a task reaches a resource only when some specific constraints are valid.

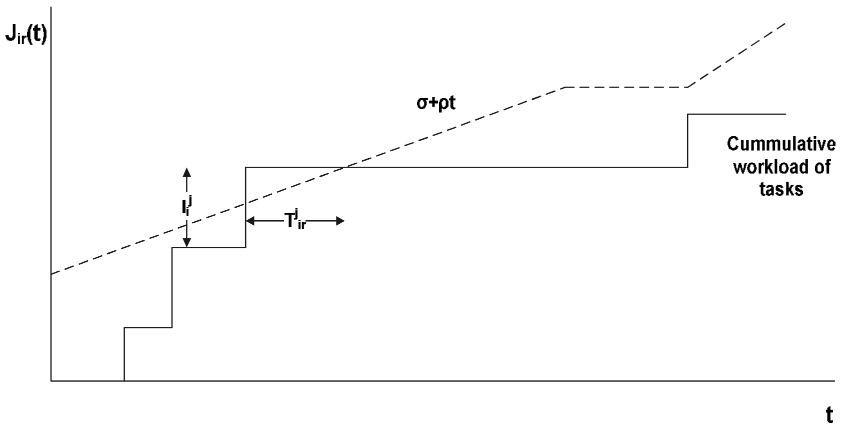

Figure 8 – The GS user is responsible for the observance of his (ρ_{ir}, σ_{ir}) constraints.

When a task is created, the GS user searches for the most suitable resource it has already registered on. We assume that task *j* of user *i* is characterized by its deadline D_i^j and its workload I_i^j (measured, e.g., in MI). In order for task *j* to be sent to resource *r* two conditions must hold. First, the task's workload must not exceed the one agreed,

$$I_i^j \leq J_{ir}^{\max} \ , \tag{11}$$

and, second, the task must not miss its deadline. One of the benefits of *(ρ, σ)* constrained GS users and of the registration phase is that the maximum delay until a task is completed on a resource can be bounded. Indeed, it can be shown, by arguing as in [15], that if conditions Eq. (8) and Eq. (10) hold and WFQ is used, then the delay a task will incur from the time it reaches resource *r* until it finishes its execution at a selected resource is at most

$$\frac{s_{ir}}{g_{ir}}+\frac{J_{ir}^{\max}}{g_{ir}}+\frac{J_{r}^{\max}}{C_{r}}, \qquad (12)$$

where g_{ir} is the minimum value of $g_{ir}(t)$ that does not invalidate Eq. (8) for any registered user. To this delay we must add the communication delay d_{ir}^{j} required for transferring the task to the resource, and the time T_{ir}^{j} the GS user withholds the task in its local queue (Figure 8). So the delay bound B_{ir}^{j} resource *r* guarantees to user *i* is

$$B_{ir}^{j} \le T_{ir}^{j} + d_{ir}^{j} + \frac{s_{ir}}{g_{ir}}+\frac{J_{ir}^{\max}}{g_{ir}}+\frac{J_{r}^{\max}}{C_{r}} \, . \qquad (13)$$

Based on Eq. (8) and assuming $w_{ir}=1$, for all *i, r* , we have:

$$B_{ir}^{j} \le T_{ir}^{j} + d_{ir}^{j} + \frac{\left(s_{ir}+J_{ir}^{\max}\right)\cdot\left(N_{r}(t)+1\right)+J_{r}^{\max}}{C_{r}} \, . \qquad (14)$$

When the GS user does not have any more tasks to submit, he can either do nothing or he can deregister from his registered resources. In the latter, dynamic, case the other GS users are informed for the user's deregistration and they can try to register to these resources.

Furthermore, we can pipeline the communication delay T_{ir}^{j} and the input queuing delay d_{ir}^{j} to obtain:

$$B_{ir}^{j} \le \max(T_{ir}^{j}, d_{ir}^{j}) + \frac{\left(s_{ir}+J_{ir}^{\max}\right)\cdot\left(N_{r}(t)+1\right)+J_{r}^{\max}}{C_{r}} \, . \qquad (15)$$

By pipelining, we mean that if d_{ir}^{j} is larger than T_{ir}^{j}, then the user *i* sends task *j* to the selected resource immediately, without waiting for the T_{ir}^{j} time period to expire, while if T_{ir}^{j} is larger than d_{ir}^{j} then the user sends the task to the resource after $T_{ir}^{j} - d_{ir}^{j}$ time. In both cases time savings are achieved.

In order for a task *j* of GS user *i* to be scheduled on resource *r*, its deadline D_{i}^{j} must be smaller than the resource's delay bound B_{ir}^{j}:

$$B_{ir}^{j} \le D_{i}^{j} \, . \qquad (16)$$

If more than one resources fulfill the conditions of Eq. (11) and Eq. (16), the GS user can choose one based on any other desired optimization criterion. If no resource fulfills these conditions, the GS user drops the task or schedules it like a BE task. Also, from Eq. (15) we conclude that it may be beneficial to partition the resources in groups offering different maximum delay guarantees. More specifically, the a priori determination of a resource's computational capacity *C*, maximum task size *J*, maximum burstiness *σ* and maximum number of GS users allowed *N*, provides a guaranteed maximum delay for the tasks sent to that resource:

$$D(C,J,N,s) \le \max(T,d) + \frac{\left(s+J\right)\cdot\left(N+1\right)+J}{C}, \qquad (17)$$

where *T* and *d* do not depend on the resource but on the user side. If *σ* is expressed as a multiple of *J*, $\sigma = m \cdot J$ (that is, the user is allowed to send up to *m* maximum-sized tasks in a very short interval if he has not sent any other tasks recently), then Eq. (15) can be rewritten as:

$$D(C,J,N,m) \le \max(T,d) + \frac{\left((m+1)\cdot N+1\right)\cdot J}{C} \quad . \tag{18}$$

3.1.2 Resources

To obtain a specific implementation of the proposed QoS framework, we distinguish four types of resources, to be referred to as GS, BE, GS_BE_EQUAL and GS_BE_PRIORITY resources. GS resources handle only tasks originating at GS users. When a GS task arrives at a GS resource, it is queued at the local WFQ scheduler. When a machine becomes free, the local WFQ scheduler selects the next GS task for execution. BE resources handle tasks originating only from BE users. The arriving tasks are placed in a queue and served following a First Come First Serve (FCFS) policy to the first available machine. GS_BE_EQUAL resources handle tasks originating from both GS and BE users. GS tasks are served using a local WFQ scheduler as in GS resources. Each arriving BE task is considered as belonging to a new user who wants to register to the resource. So a BE task is queued in the local WFQ scheduler only if the condition of Eq. (8) holds for all the registered users. In this case, the number of registered users is increased by one and when the BE task finishes execution it is decreased by one. If Eq. (8) is violated for at least one registered user then the task is rejected and a failure notice is returned to the originating user. GS_BE_PRIORITY resources handle both GS and BE tasks, but not in the same way. GS tasks are handled by the local WFQ scheduler, while BE tasks are placed in a FCFS queue. When a machine becomes free, the tasks in the local WFQ scheduler are handled first. If there are no such tasks, the BE tasks from the FCFS queue are served. A GS_BE_PRIORITY resource is characterized as preemptive if upon the arrival of a GS task, a BS task currently under execution is paused and replaced by the new GS task; otherwise, the GS_BE_PRIORITY resource is characterized as non-preemptive. Finally, a BE task is scheduled to a GS_BE_EQUAL or GS_BE_PRIORITY resource only when its size is smaller than the resource's maximum acceptable task size.

When a GS_BE_PRIORITY non-preemptive resource is used, the delay bound for GS tasks of Eq. (13), becomes

$$B_{ir}^{j} \le T_{ir}^{j} + d_{ir}^{j} + \frac{s_{ir}}{g_{ir}} + \frac{J_{ir}^{\max}}{g_{ir}} + \frac{J_{r}^{\max}}{C_r} + R_r \tag{19}$$

where R_r is the residual time for the BE task found at the resource (if any) to complete execution:

$$R_r \le \frac{J_r^{\max}}{C_r} \tag{20}$$

In all other resource types (namely, GS, GS_BE_PRIORITY preemptive) R_r equals to 0.

Therefore, delay bounds are provided to GS tasks submitted to GS, GS_BE_EQUAL or GS_BE_PRIORITY resources, while fairness is also provided among BE users for tasks submitted to BE or GS_BE_PRIORITY resources. Table 6 presents the user/resource combinations that provide delay bounds.

Resource	Delay Bound for GS users
GS	$\max\left(T_{ir}^{j}, d_{ir}^{j}\right) + \frac{\left(s_{ir} + J_{ir}^{\max}\right)\cdot\left(N_r(t)+1\right) + J_r^{\max}}{C_r}$
BE	-
GS_BE_EQUAL	$\max\left(T_{ir}^{j}, d_{ir}^{j}\right) + \frac{\left(s_{ir} + J_{ir}^{\max}\right)\cdot\left(N_r(t)+1\right) + J_r^{\max}}{C_r}$

GS_BE_PRIORITY preemptive	$\max\left(T_{ir}^{j}, d_{ir}^{j}\right)+\frac{\left(s_{ir}+J_{ir}^{\max}\right)\cdot\left(N_r(t)+1\right)+J_r^{\max}}{C_r}$
GS_BE_PRIORITY non-preemptive	$\max\left(T_{ir}^{j}, d_{ir}^{j}\right)+\frac{\left(s_{ir}+J_{ir}^{\max}\right)\cdot\left(N_r(t)+1\right)+2\cdot J_r^{\max}}{C_r}$

Table 6: Delay bounds given to GS users for each resource type.

3.2 Extensions of the Proposed Framework

3.2.1 Distributed, Centralized and Hybrid Implementations

In Section 3.1 we assumed a distributed implementation of our proposed QoS framework, where registration is done by each user (or VO) by communicating directly with the resource and negotiating its *(ρ, σ)* constraints. However, other approaches can also be used.

In the centralized approach the registration of GS users to resources is handled by a central meta-scheduler. The meta-scheduler accepts, from the GS users, registration requests containing their requested *(ρ, σ)* parameters, and searches for resources that can satisfy these constraints. The meta-scheduler is responsible for enforcing the *(ρ, σ)* constraints of the GS users. The centralized approach has various advantages. First, many of the heavy operations are transferred to a central, possibly more powerful, machine. Second, it is possible to use more than one central meta-scheduler to balance the load and the traffic in the Grid Network. On the other hand, the use of a single central meta-scheduler increases the risk of a failure in the Grid Network. Also GS task average total delay increases, because of the delay induced by the communication between the GS users and the central meta-scheduler.

A hybrid approach is also possible, where a meta-scheduler is responsible for the registration of GS users to resources, but following the registration, the users submit their tasks directly to one of their registered resources, and are themselves responsible for the observance of their *(ρ, σ)* constraints. Using this hybrid approach the meta-scheduler is relieved from the burden of scheduling GS tasks. Furthermore, GS tasks do not experience the delay of communicating with the meta-scheduler, reducing the total task delay.

3.2.2 Multi-machine Resources

The proposed framework can easily be extended to the case of resources that consist of many machines-CPUs, provided that some of the definitions and conditions given earlier are appropriately modified. The total computational capacity C'_r of a multi-machine resource *r* is expressed as:

$$C'_r=\sum_{j=1}^{M_r} C_{rj} \qquad (21)$$

where C_{rj} is the computational capacity of machine *j* and M_r is the number of machines (CPUs) in resource *r*. However, in the multi-machine resources case the term C_r used in Eq. (8) and in Eq. (13) is not always equal to C'_r. Furthermore, we assume that the local scheduler assigns tasks to the first available machine-CPU, in a round-robin manner.

In Eq. (8), $g_{ir}(t)$ is the average service rate resource *r* guarantees to provide to user *i*. Since C'_r is the total service rate the user has access to from resource *r*, C_r in Eq. (8) must be replaced by C'_r, yielding

$$r_{ir} \leq g_{ir}(t) = \frac{w_{ir} \cdot \sum_{j=1}^{M_r} C_{rj}}{\sum_{k=1}^{N_r(t)+1} w_{kr}} \qquad (22)$$

Since tasks are non-divisible, the resource cannot use its total computational capacity to process a task. The worst case is obtained when a task is assigned to the machine (CPU) with the lowest computational capacity $C_r^{\min} = \min_J C_{rj}$. Therefore, C_r in Eq. (13) and in all the other delay bounds given in Section 3.1 has to be replaced by $C_r^{\min}$. For example, Eq. (13) becomes:

$$B_{ir}^j \leq T_{ir}^j + d_{ir}^j + \frac{s_{ir}}{g_{ir}} + \frac{J_{ir}^{\max}}{g_{ir}} + \frac{J_r^{\max}}{C_r^{\min}} \qquad (23)$$

3.2.3 Scheduling Without A-priori Knowledge of the Task Workloads

The proposed QoS framework offers hard delay guarantees to GS users that respect their negotiated *(ρ, σ)* constraints. The observance of a GS user's *(ρ, σ)* constraints requires the a-priori knowledge or accurate estimation of the task workloads, which is not always possible. We assume that a GS user chooses his *(ρ, σ)* constraints based on past statistics and approximate assumptions about his task generation rate and workload. Even though it is clearly possible for the user to measure and control dynamically the rate at which he submits tasks to the Grid, it may not be easy to measure and control their workload.

In what follows we examine how our framework can be extended to operate without a-priori knowledge of the task workloads. Specifically, we propose the following methods:

- Conservative Task Submission: The meta-scheduler assumes that each new task has workload equal to the user's maximum task workload, which is the maximum of $J_{ir}^{\max}$ for all the user's *i* registered resources *r*, and updates the variable $J_{ir}(t)$ based on this assumption. In case Eq. (10) is violated, the corresponding task is backlogged.
- *n*-Window Aggressive Task Submission: The meta-scheduler schedules up to *n* consecutive new tasks of GS user *i* to a resource *r*, assuming their workload is equal to zero. Any new task *j* destined for resource *r* that arrives after the *n* consecutive tasks is backlogged, until a workload feedback message for any of these *n* tasks arrives from resource *r*. Specifically, when a task completes execution on resource *r*, the resource informs the meta-scheduler of the task's actual workload, and the variable $J_{ir}(t)$ is updated. When no limit is placed on the number *n* of consecutive tasks that can be sent without a priori knowledge of their workload, then no deterministic delay bound can be given. We refer to this case as Full Aggressive Task Submission method.
- Conservative Task Submission with Feedback: The above two methods can be combined. The meta-scheduler schedules new tasks of GS user *i* to resource *r*, assuming that their workload is equal to the user's maximum task workload, and updates $J_{ir}(t)$ based on this assumption. When a task completes execution, a workload update message is sent back to the meta-scheduler, which corrects its previous assumption on the task workload and updates $J_{ir}(t)$ accordingly. In case Eq. (10) is violated, the corresponding task is backlogged.

3.2.4 (*ρ*, *σ*) Constraints Selection Policy

To determine the exact values of the *(ρ, σ)* parameters between a user (or VO) and a resource, each user has to estimate his average long term service requirements, and its desired burstiness (or the delay he can afford for "smoothening" the traffic to fit a given *σ*). It is natural to assume that the price a user pays for the use of the Grid is an increasing function of *ρ* and *σ*. If the user chooses *ρ* too close to his average long term needs, or chooses a small *σ*, then an arriving task may have to suffer a long delay *T* waiting for Eq. (10) to become valid. If the user can afford to pay a higher price, it may be beneficial to overestimate *ρ* or *σ* so as to reduce this delay. The *(ρ, σ)* parameters requested by the user may be too large for the meta-scheduler to accept. During the registration phase the meta-scheduler will determine the exact values of these parameters, based on the computational power of the resource, the distance from the user, the resource's delay bound, the number of users already registered, etc.

3.3 Performance Results

We implemented the centralized version of the proposed QoS framework in the GridSim simulator [25]. For a GS user, his *(ρ, σ)* constraints are specified, along with the maximum workload *J* of his generated tasks. Each resource is of a specific type and has a maximum acceptable task workload. The central meta-scheduler is responsible for the registration phase, the observance of the *(ρ, σ)* agreements between GS users and resources, and the assignment of tasks to resources. All users register to resources at the beginning of the simulation and remain registered for its entire duration. The local-scheduler of a resource is equipped with a FIFO queue and a Self-Clocked Fair Queueing (SCFQ) scheduler. SCFQ is a variation of Weighted Fair Queueing (WFQ) that is easier to implement than WFQ. Based on their type and the type of the resources, tasks are assigned either to the FIFO queue or to the SCFQ scheduler. We assume that the local-scheduler uses a space-sharing resource allocation policy. We also assume, unless stated otherwise, that each resource consists of a single-CPU machine and task workloads are known a-priori.

In order to obtain realistic simulation parameters, we used the results of the Grid profiling study of [24], where numeric data (as well as analytic models) on the cumulative distribution functions, average values and standard deviations of the task inter-arrival times, queue waiting times, task execution times, and data sizes exchanged at the *kallisto.hellasgrid.gr* cluster (part of the EGEE infrastructure) were presented.

Based on these numeric data we simulated three GS users, named U1, U2 and U3, corresponding to three of the five VOs presented in [24] (the Atlas, Magic and Dteam VOs). Using the VOs average task execution times and processor speed we calculated their corresponding average task workloads, measured in Million Instructions (MI). We also drew information from [24] about the task inter-arrival times. We then calculated the *(ρ, σ)* constraints of each GS user. Specifically, the *ρ* parameter of a GS user is calculated by dividing its average task workload by its average task inter-arrival time, while the *σ* parameter is selected to be *m=5* times larger than the GS user's average task workload. The simulation parameters are shown in Table 7.

User	Task Workload	Task Inter-arrival	ρ	σ
Atlas/U1	Fixed: 419900000 MI	Fixed: 546 secs/task (s/t)	Fixed: 768473 MIPS	Fixed: 2099500000 MI
Magic/U2	Fixed: 71383000 MI	Fixed: 3278 secs/task (s/t)	Fixed: 21773 MIPS	Fixed: 356915000 MI
Dteam/U3	Fixed: 419900 MI	Fixed: 6010 secs/task (s/t)	Fixed: 699 MIPS	Fixed: 2099500 MI

Table 7: GS users task workload and inter-arrival time.

In our simulations we also included two BE users, named U4 and U5. U4's average task inter-arrival times change in every simulation, while U5's remain the same (Table 8). The task workloads submitted by these users were equal to the Atlas/U1 average task workload (namely, 10000 MI).

User	Task Workload	Task Inter-arrival
U4	419900000 MI	Fixed: 5464, 4371, 3278, 2186, 1366, 1093, 820, 546, 55 secs/task (s/t)
U5	419900000 MI	Fixed: 5464 secs/task (s/t)

Table 8: BE users task workloads and inter-arrival time.

At the *kallisto.hellasgrid.gr* cluster presented in [24], 60 working nodes (60 Intel Xeon processors) are used, with a total capacity of 60*26000 MIPS. In our simulations, we used 3 clusters (resources) each having one machine with one CPU of computational capacity equal to a multiple of 26000 MIPS. More specifically, we assumed that resources R1, R2 and R3 have 30, 20, and 10 CPUs, which corresponds to a total computation capacity of 780.000 MIPS, 520.000 MIPS and 260.000, respectively. The resource type scenarios examined are presented in Table 9. For example, in the GB scenario of Table 9, resource R1 and R2 are allocated for serving GS users, while resource R3 serves BE users. When the BE resource scenario is used then all the GS users (U1, U2 ,U3) are treated as BE users with the same characteristics as before.

The meta-scheduler uses a two-phase (task-ordering and task-to-resource assignment) procedure for scheduling BE users, with task collection period equal to 1 second. Unless stated otherwise, the two-phase scheduling procedure uses the Earliest Deadline First (EDF) algorithm for the ordering phase and the Earliest Start Time (EST) algorithm for the assignment phase. All the users have non-critical deadlines, with values equal to 110 seconds. In general, if a critical deadline expires, the corresponding task is removed from the Grid, while if a non-critical deadline expires the task remains in the Grid, but is recorded as a deadline miss. The deadline's value was selected based on Eq. (12) by adding a small time overhead to account for the input queueing (T) and communication (d) delays. Furthermore, in our simulations we assumed that the resources, the users and the central meta-scheduler communicate directly with each other over links of equal bandwidth and zero propagation delay. The resources use a space-sharing policy, and their maximum acceptable task workload is taken to be larger than the task workload produced by any user. The GS users maximum task workload is equal to their corresponding fixed task workload (Table 7). Finally, the sizes of the data sent to a resource from a user before task execution and the sizes of the data produced by a resource when the task is completed are the same for all users and equal to 1000 bytes.

Scenarios	**R1**	**R2**	**R3**
GB	GS	GS	BE
GBE	GS_BE_EQUAL	GS_BE_EQUAL	BE
GBP	GS_BE_PRIORITY non-preemptive	GS_BE_PRIORITY non-preemptive	BE
BE	BE	BE	BE

Table 9: Resources scenarios.

To asses the performance of the proposed framework we used the following metrics:

- The per user percentage of the number of tasks that miss their non-critical deadlines over the total number of tasks the user creates.
- The resource use, defined as the total time a resource is used for the execution of tasks.
- The per user percentage of the number of failed BE tasks over the total number of tasks the user creates. A BE task fails (and is dropped) when it arrives at a GS_BE_EQUAL resource and finds that it cannot be scheduled without violating the delay guarantees given to the already registered GS users.
- The percentage of GS tasks that have to wait at the input (backlogged), in order for the GS user to remain (ρ, σ) constrained, over the total number of GS tasks.

3.3.1 Results Obtained

A number of simulations were conducted to validate that the proposed framework indeed provides hard delay guarantees to GS users and fairness to BE users. In our simulations we used 3 GS and 2 BE users, 3 single-machine and single-CPU resources, and a meta-scheduler. We examine all the resource scenarios presented in Table 9. The task workloads and inter-arrival times follow a fixed (deterministic) distribution, using the values of Table 7 and Table 8, respectively. The task inter-arrival times (task generation rates) of user U4 change in many simulation scenarios. Using these parameters in our simulations, we observe that the GS users register to one resource; specifically, U1 registers to R1 and U2 and U3 to R2.

3.3.1.1 *Guaranteed Delay Bounds for GS Users*

Figure 9.a shows that our scheme succeeds in providing hard delay guarantees to the GS users. Figure 9.a presents the per user percentage of tasks that miss their non-critical deadline. This percentage is presented for all resource scenarios (GB, GBE, GBP, BE) and for different values for the task inter-arrival times of BE user U4 (1093, 546, 55 secs/task). First, and most importantly, we observe that in all cases the GS users (U1, U2, U3) do not miss any of their deadlines, verifying that our framework succeeds in providing hard delay guarantees such GS users. Only when the BE resource scenario is used, where GS users are treated as BE users, do users start missing many of their deadlines.

In the GBE and the GBP scenarios (Table 9) fewer tasks miss their deadlines, but in the GBE resource scenario many BE tasks fail (Figure 9.b). So the GBP resource scenario, where resource R1 and R2 are used by both GS and BE users but with different priorities, seems to be the best in terms of the number of tasks successfully scheduled without missing their deadlines. This is because in this resource scenario better use of the available resources is achieved, by multiplexing GS and BE users (with different priorities) on resources R1 and R2.

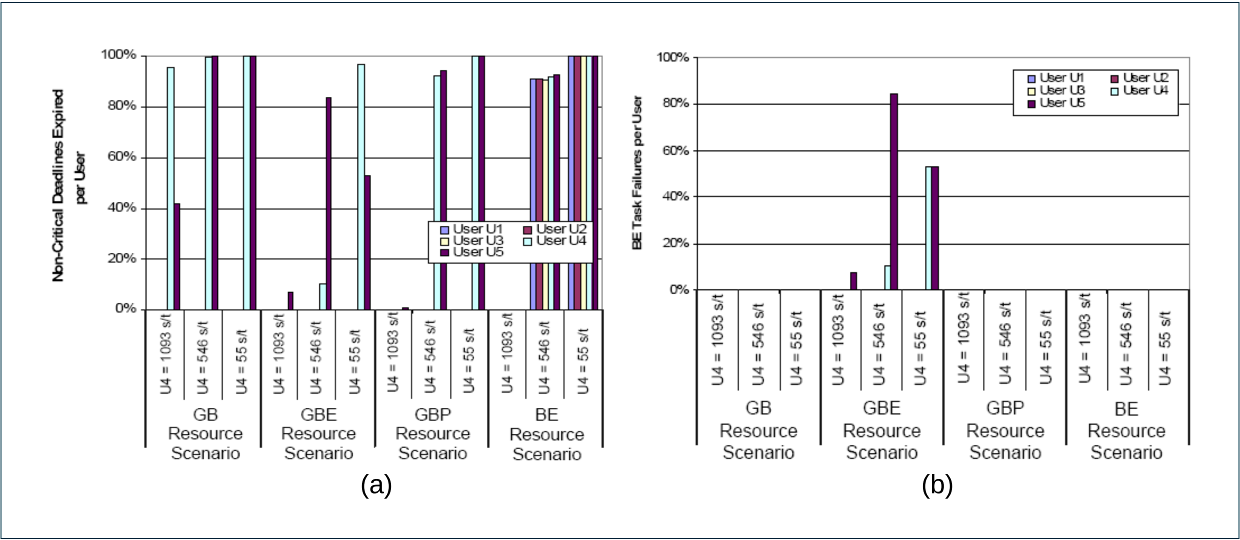

Figure 9 – The per user percentage of (a) the number of tasks that miss their non-critical deadlines, (b) the number of failed tasks, for various resource scenarios and task inter-arrival times (in secs) of BE user U4.

In Figure 10.a the total time each resource is used is presented for the same scenarios as before. Resource R3 is utilized more in the GB resource scenario, since it handles exclusively BE tasks. In the other resource scenarios, all resources can serve both GS and BE tasks and as a result the use of resource R3 is smaller. Finally, in Figure 10.b the standard deviations of the resources use are presented. The standard deviation is high in the GB scenario, where resource R3 is more utilized than resources R1 and R2, while it is very small for the GBP scenario. This indicates that the GBP scenario makes more efficient and uniform use of the available resources than the other scenarios.

We also observed that the total task delays of the GS users have very small deviations, compared to the total task delay deviations of the BE users. The tasks of GS user U1 have a smaller total task delay than the equally-sized tasks of BE users U4 and U5. Also, the total task delays of users U4 and U5 are larger in the GS scenario than in the other scenarios, because in the first case only one resource is available for BE users. The total task delays and the corresponding standard deviations for all the users, increase as expected when the task inter-arrival rate of user U4 increases. Finally, because the generated GS user tasks conform to the agreed *(ρ, σ)* constraint none of their tasks is ever backlogged at the input of the Grid.

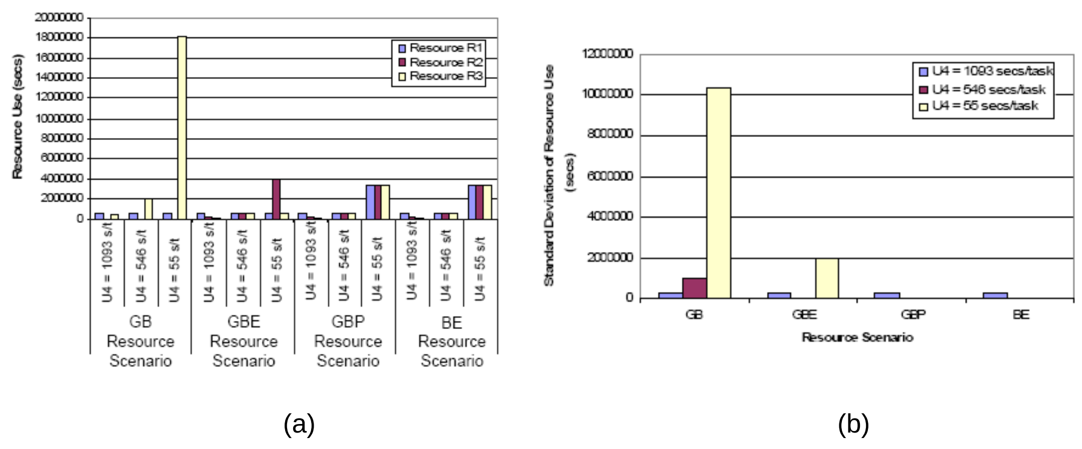

Figure 10 – (a) The resource use, (b) the standard deviation of the resource use, for various resource scenarios and task inter-arrival times of BE user U4.

3.3.1.2 *(ρ, σ) Constraints Violation*

Next, we look at the behavior of the proposed schemes when the GS users violate their *(ρ, σ)* agreements made with the Grid Network. In theory when a GS user starts violating his *(ρ, σ)* constraints, then either his GS tasks are backlogged until Eq. (10) becomes valid again, or some of his GS tasks are handled by our framework as BE tasks because there is no GS capable resource that can handle them before their deadline expires. In the first case we expect that the task total delay of GS tasks will increase. In the second case we expect that many of the GS tasks will miss their deadlines.

In addition to the fixed (deterministic) distribution we also considered in our simulations other distributions for the task inter-arrival times of GS users U1, U2 and U3. Specifically, we obtained results for the following distributions of the task inter-arrival times:

- uniform distribution with minimum value 5%, 20%, or 50% smaller than the corresponding fixed value of Table 7 and maximum value 5%, 20%, or 50% larger than the corresponding fixed value of Table 7, to be referred as Un.5, Un.20, Un.50 distributions, respectively.
- uniform distribution with minimum value 50% smaller than the corresponding fixed value of Table 7 and maximum value 20% larger than the corresponding fixed value of Table 7, to be referred as Mixed distribution.

Figure 11.a shows the per user percentage of tasks that miss their non-critical deadline in the GBP resource scenario (Table 9), under various distribution scenarios (Fixed, Un.5, Un.20, Un.50, Mixed), and for mean inter-arrival times of 1093, 546 and 55 secs/task for user U4. We observe that in the Un.5 and Un.20 distribution scenarios the number of non-critical deadlines expired for all the users is almost the same with that of the original fixed distribution scenario. However, our results indicate a small rise in the number of GS tasks that are backlogged and a corresponding increase in their average delay. On the other hand in the Un.50 and the Mixed distribution scenarios the number of non-critical deadlines expired increases, and, more importantly, the GS users miss many of their non-critical deadlines. This happens because these GS tasks cannot be served by any GS capable resource before their deadline expires, and as a result are handled by our framework as BE tasks without any delay guarantees. From these results we conclude that as long as the GS users respect their *(ρ, σ)* constraints, even with small deviation, our QoS framework succeeds in providing them with hard delay guarantees.

Figure 11.b shows the per user percentage of tasks that miss their non-critical deadline for the GBP and BE resource scenarios, under the Mixed distribution scenario. We observe that though the GS users miss many of their non-critical deadlines the use of the proposed framework (under the GBP resource scenario) benefits in most cases the users, in terms of non-critical deadlines missed, than when our QoS framework is not used (that is, under the BE resource scenario).

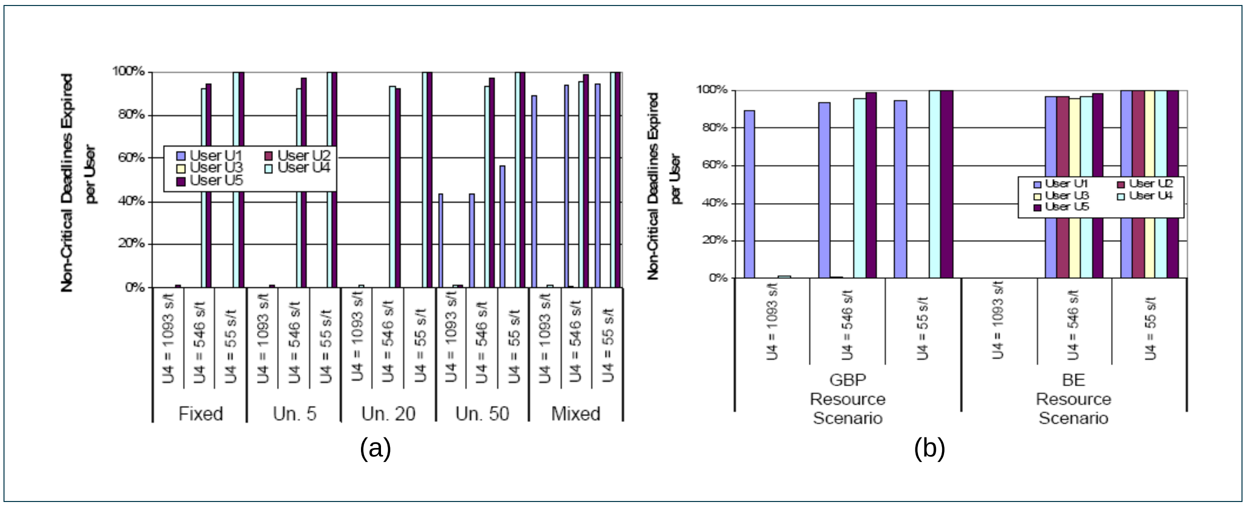

Figure 11 – The per user percentage of the number of tasks that miss their non-critical deadlines using the GBP resource scenario, (a) for various task inter-arrival times (secs/task) of GS users U1, U2, U3 (Fixed, Un.5, Un.20, Un.50, Mixed) and of BE user U4, (b) , for various resources scenarios and task inter-arrival times (in secs/task) of BE user U4. We assume inter-arrival times that follow the Mixed distribution for the GS users.

4 Joint Communication and Computation Task Scheduling in Grids

This section addresses a specific instance of the general problem of co-allocation of Grid resources. Resource co-allocation is one of the most challenging problems in Grids. The co-allocation problem for computational Grids has been defined in [27]. In the Condor project, the gang matchmaking scheme [26] extends the matchmaking model in order to support the co-allocation of resources.

In order to co-allocate resources, the scheduler has to orchestrate resources belonging to different sites and administrative domains. To do so, advance reservation of these resources has to be supported by the local resource management systems. Thus, advance reservations is a mechanism to provide end-to-end Quality of Service to the Grid users. The Globus Architecture for Reservation and Allocation (GARA) [11] is a framework for advance reservations that treats in a uniform way various types of resources such as communication, computation, and storage. Although GARA has gained popularity in the Grid community, its limitations in coping with current application requirements and technologies led to the proposal of the Grid Quality of Service Management (G-QoSm) framework. The WS-Agreement protocol [28] proposed by the GRAAP working group in the Open Grid Forum (OGF) can be used as a negotiation protocol. General Grid workflow management systems have been developed by several projects: Condor DAGman [30], GridFlow [31], Gridbus [32], and UNICORE [51]. A taxonomy for workflow management systems in Grid Computing is presented in [29].

Grid applications can be categorized as CPU-intensive, data-intensive applications, or both. However, almost all tasks have a computation and a communication part, even if one part is negligible. For example, it is usual for a task to require the movement of a large chunk of data from the location of the user or a data repository site to the computation resource where the task will be executed. In this section we address such a problem, which can be also viewed as a simple form of workflow with two successive steps: a communication and a computation resource reservation.

Some types of joint communication and computation problems have been examined. In [33] the authors decoupled the data replication and computation problems and evaluated the performance of data and task schedulers working in a cooperatively manner. In [34] the proposed scheduler selects the computation resource to execute a task based on the computation resource capability, the bandwidth available from the data hosting site to the computation resource and the cost of the data transfer. Similar algorithms have been examined in multimedia networks where the co-allocation of computation, communication (bandwidth) and other resources are examined [35]. The authors in [36] introduced the concept of scheduling and routing of advance reservation requests in the communication plane. More specifically, in [36] several algorithms for advance reservations are proposed, in which the starting time of the communication reservation is specified or is flexible. The complexity of these algorithms is also discussed.

In this section, we propose a multicost algorithm that jointly addresses a type of communication and computation scheduling problem [56]. In comparison to the solutions proposed in [33],[34] and[35], our algorithm also uses advance reservations for the time scheduling of the communication resources, in a similar way to [36]. Multicost algorithms have mainly been used for QoS routing problems. In [47] the authors proved that QoS routing with parameters being the bandwidth and the delay is not NP-complete. The general Multiconstrained Path Problem (MCP) is discussed in [46]. The present work is the first time a multicost algorithm is used for the joint communication and computation problem in a Grid environment. Moreover, a key difference to other multicost approaches is that the proposed multicost algorithm is designed to cope with the time scheduling of the resources, utilizing temporal information to perform advance reservations.

We assume that task processing consists of two successive steps: (i) the transfer of data from the scheduler or a data repository site, which we will call source, to the cluster or computation resource (these terms will be used interchangeably in this section) in the form of a connection or a data burst and (ii) the execution of the task at the cluster. The link utilization profiles, the link propagation delays, the cluster utilization profiles and the task parameters (input data size, computation workload and maximum acceptable delay) form the inputs to the algorithm. The proposed multicost algorithm selects the cluster to execute the task, determines the path to route the input data, and finds the starting times for the data transmission and the task execution at the cluster, performing advance reservations. The algorithm takes its decisions based on the resources (link and cluster) utilization information available at the scheduler when the algorithm is executed. Note that the proposed algorithm is designed for a distributed architecture and thus the information maintained at the scheduler can be outdated, but the algorithm can be easily extended to function in a centralized manner.

The proposed algorithm consists of three phases: it first uses a multicost algorithm to compute a set of candidate non-dominated paths from the source (scheduler or data repository site) to all network nodes (clusters or not). Secondly, the algorithm obtains the set of candidate non-dominated (path, cluster) pairs from the source to all clusters that can process the task. Finally, the algorithm chooses from the previously computed set a pair that minimizes the task completion time, or some other performance criterion. An important drawback of the algorithm outlined above is that in its first phase the number of non-dominated paths may be exponential. To obtain a polynomial-time heuristic algorithm, we use a pseudo-domination relationship between paths to prune the solution space.

We evaluate the performance of the optimal task routing and scheduling algorithm and of its proposed polynomial-time heuristic variation using network simulation experiments, and compare it to that of algorithms that handle the computation or communication part of the problem separately. Our results indicate that when the tasks are CPU- and data- intensive it is beneficial for the scheduling algorithm to jointly consider the computational and communicational problems, as our proposed algorithms do. Comparing the optimal multicost algorithm to the proposed heuristic we observe that they exhibit similar performance. Thus the proposed heuristic combines the strength of the optimal algorithm with a low computation complexity.

In Section 4.1 we present the utilization profiles of the communication and computation resources. In Section 4.2 we formally define the problem and show how to compute the utilization profile of a path and the utilization profile of a cluster over a path based on the profiles defined in Section 4.1. The joint communication and computation scheduling algorithm is presented in Section 4.3 and its heuristic variation in Section 4.4. Section 4.5 presents performance results.

4.1 Communication and Computation Utilization Profiles

4.1.1 Link Utilization Profiles

In a network that uses advance reservations, each node needs to keep a record of the capacity reserved on its outgoing links, as a function of time, in order to perform channel scheduling and reservation [45]. Assuming each connection reserves bandwidth equal to *r* for a given time duration, the utilization profile U_l(t) of a link *l* is a stepwise binary function with discontinuities at the points where reservations begin or end, and is updated dynamically with the admission of each new connection. We define the capacity availability profile of link *l* of capacity C_l as $C_l(t)=C_l-U_l(t)$. To obtain a data structure that is easier to handle in an algorithm, we discretize the time axis in steps (timeslots) of duration τ_l and define the *binary r-capacity availability vector* $\hat{C}_l(r)$, abbreviated CAV, as the vector whose *k*-th entry is:

$$\left\{\hat{C}_l(r)\right\}_k = \begin{cases} 1, \text{ if } C_l - U_l(t) > r \\ 0, \quad \text{othewise} \end{cases}, \text{for all } (k-1)\cdot t_l \le t \le k\cdot t_l, k=1,2,...,d_l$$

where d_l the dimension of the CAV (see Figure 12).

The data structures defined above can be useful in a number of network settings. For example, in an optical WDM network with full wavelength conversion and *w* wavelengths per link, each of capacity C_w, the capacity of a link *l* is $C_l=w\cdot C_w$. A connection that wants to reserve *k* wavelengths, $1\le k \le w$, has requested rate $r = k\cdot C_w$. If we can find a path of available capacity at least *r* then the connection can be established (full wavelength conversion). If no wavelength converters are available, each link needs to keep track of the utilization profile of each of its *w* wavelengths separately. Thus, a node has to maintain *w* binary (a wavelength can be reserved or not for a given time) utilization profiles for each outgoing link. In this case the network can be viewed as *w* “parallel” networks, each having a single wavelength. In order to simplify notation, in the remainder of the section and when no confusion can arise, we will denote the profiles and $\hat{C}_l(r)$ of a link *l* by $\hat{C}$, suppressing the dependence on *l* and *r*.

In what follows we assume that the network connecting the clusters, schedulers and data repositories follows the Optical Burst Switching (OBS) paradigm [43]. Note that this assumption does not limit the applicability of our algorithms, which can be used in any network supporting advance reservations. In OBS networks, the data exchanged are transmitted as data bursts that are switched though the network using a single label. This reduces the switching and processing requirements in the core network. The Grid Optical Bursts Switched (GOBS) solution has been proposed to the Open Grid Forum (OGF) as a candidate network infrastructure to support dynamic and interactive services [44].

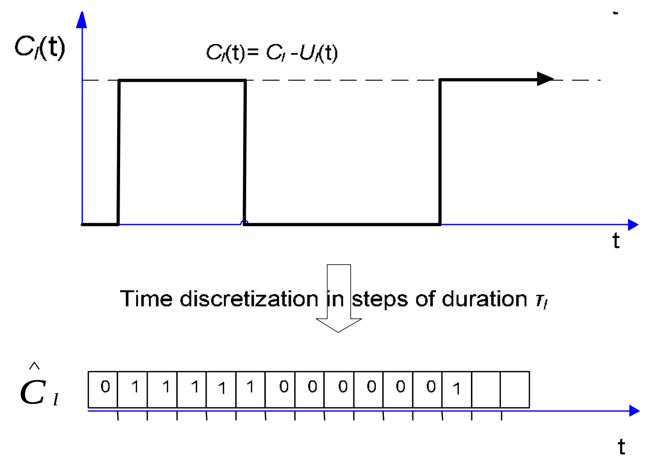
C_l(t)
C_l(t)= C_l -U_l(t)
t
Time discretization in steps of duration τ_l
$\hat{C}_l$
0 1 1 1 1 1 0 0 0 0 0 0 1
t

Figure 12: The capacity availability profile $C_l(t)$, and the binary capacity availability vector $\hat{C}_l$ of a link *l* of capacity C_l.

4.1.2 Clusters Utilization Profiles

We assume that a cluster-site *m* consists of W_m CPUs of equal processor speed C_m (measured, e.g., in MIPS). We also assume that when a task starts executing at a CPU it cannot be preempted. A task requests to be executed in *r* CPUs ($r \leq W_m$), and can be scheduled for execution in the future.

The utilization profile $U_m(t)$ of cluster *m* is defined as an integer function of time, which records the number of processing elements that have been committed to tasks at time *t* relative to the present time. The maximum value of $U_m(t)$ is the number of CPUs W_m, and it has a stepwise character with discontinuities of height *r* (always integer number) at the starting and ending times of tasks. In case all tasks request a single CPU the steps are always unitary. We defined the cluster availability profile, which gives the number of CPUs that are free as a function of time, as $W_m(t)= W_m – U_m(t)$. In order to obtain a data structure that is easier to communicate and store we discretize the time axis in steps of duration τ_m and defined the binary *r*-cluster availability vector $\hat{W}_m(r)$, as follows:

$$\left\{\hat{W}_m(r)\right\}_k = \left\{\begin{array}{ll}1, \text{ if } W_m - U_m(t) > r \\ 0, & \text{othewise}\end{array}\right\}, \text{ for all } (k-1)\cdot t_m \leq t \leq k\cdot t_m, k=1,2,...,d_m$$

where d_m is the maximum size of the $\hat{W}_m(r)$ vector (see Figure 13). To simplify presentation, we assume for this study that each task requests *r*=1 CPUs, which is the most usual case. Then, we can denote $\hat{W}_m(r)$ by $\hat{W}_m$ suppressing the dependence on *r*.

The discretization of the time axis results in some loss of information, and provides a tradeoff between the accuracy and the size of the maintained information. The discretization steps τ_l and τ_m used in the link and cluster utilization profiles, respectively, can be different to account for the different time scales in the reservations performed in these resources and to separately control the efficiency-accuracy we want to obtain in each case. Note that the timeslot-based management of Grid resources is an established and efficient way to manage utilization information [48]. This approach is already used in different environments, e.g. denoted as *timeslot table* in GARA [11], and used in the VIOLA testbed [49], [50].

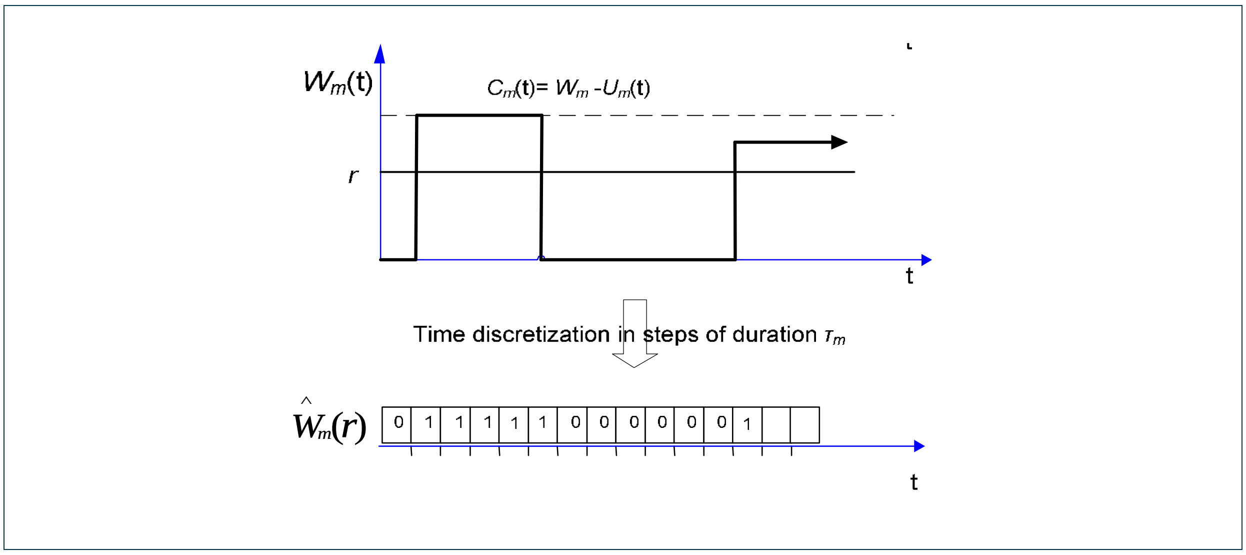
W_m(t)
C_m(t)= W_m -U_m(t)
r
t
Time discretization in steps of duration τ_m
$\hat{W}_m(r)$
0 1 1 1 1 1 0 0 0 0 0 0 1
t

Figure 13: The cluster availability profile $W_m(t)$, and the binary *r*-cluster availability vector $\hat{W}_m(r)$ of a cluster *m*.

4.1.3 Utilization Profiles in a Distributed Architecture

In a distributed architecture, each distributed scheduler maintains a “picture” of the utilization of all communication and computation resources. In our approach, this is done by maintaining a utilization database with link and cluster availability vectors for all the resources. This picture can be different among the distributed schedulers, due to non-zero propagation delays. Update information (in the form of messages) is communicated to synchronize the locally maintained profiles with the actual utilization. Note that in the case of a centralized architecture only the single central scheduler would have to maintain such utilization information, simplifying in this way the synchronization process. Although the design of a distributed algorithm is more complex, compared to the design of a centralized scheme, a distributed algorithm is more general and applicable to more cases, since it scales better and avoids many innate drawbacks of a centralized architecture.

4.2 The Task Routing and Scheduling Problem under Consideration

We are given a Grid infrastructure consisting of a network with links *l* of known propagation delays d_l, and capacity C_l and a set *M* of clusters. Cluster $m \in M$ has W_m CPUs of a given processor speed C_m (e.g., in MIPS). A task is created by a user with specific needs: input data size *I* (bits) and computational workload *W* (MIs). The user communicates this information to its attached distributed scheduler *S* [37]. We assume that the input data are forwarded by the user to the scheduler *S* or are located at a data repository site *R*. Also, *S* has (possibly outdated) information about the capacity availability vectors $\hat{C}_l$ of all links *l*, and the cluster-availability vectors $\hat{W}_m$ of all clusters *m*. We assume that there is an upper bound *D* on the maximum delay tasks can tolerate. Even when no limit *D* is given, we still assume that the dimension d_l and d_m of the link and cluster utilization vectors are finite, corresponding to the latest time (relative to the present time) for which reservations have been made. Given the previous information, we want to find a suitable cluster to execute the task, a feasible path over which to route the data, and the time at which the task should start transmission (from the source) and execution (at the cluster), so as to optimize some performance criterion, such as the completion time of the task. In other words we want to find a (path, cluster) pair and the corresponding Time Offsets, to transmit the data of the task (TO_{path}), and execute the task at the cluster ($TO_{cluster}$). Figure 14 presents an instance of the problem.

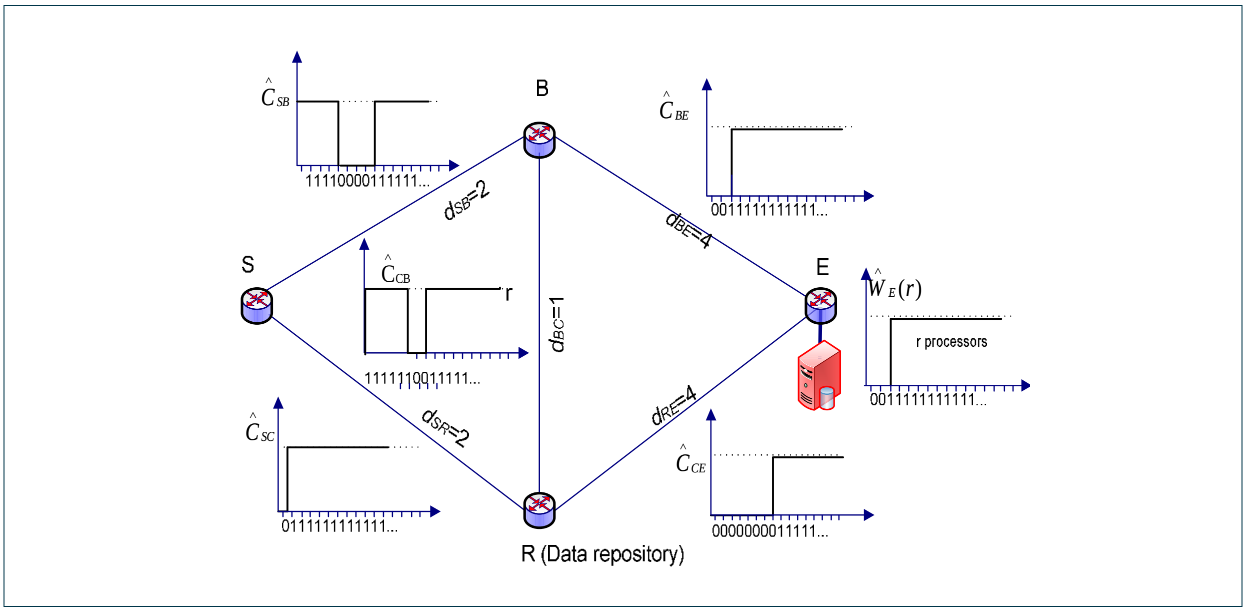

Figure 14: A task request is forwarded to the distributed scheduler *S*. The task requires the transmission of a burst with duration $b=I/C_l$ from source (which can be *S* or a data repository site *R*) and *r* CPUs to execute. Each link is characterized by its propagation delay (in τ_l time units) and its binary capacity availability vector. Node *E* has a cluster with binary *r*-cluster availability vector $\hat{W}_E(r)$.

4.2.1 Binary Capacity Availability Vector of a Path

Assuming the routing and scheduling decision is made at the distributed scheduler *S*, the capacity availability vectors of all links should be gathered continuously at *S*. For this and Section 4.2.2, we assume that the input data are located at *S*. If the input data were located at a data repository site *R*, *S* would have to compute the paths and binary cluster availability vectors over those paths (Section 4.2.2) starting from *R*.

To calculate the CAV of a path we have to combine the CAVs of the links that comprise it, as described in [38]. For example, for the topology of Figure 14, the CAV of path p_{SBE}, consisting of links *SB* and *BE*, is:

$$\hat{C}_{SBE} = \hat{C}_{SB} \oplus \hat{C}_{BE} = \hat{C}_{SB} \,\&\, \mathrm{LSH}_{2\cdot d_{SB}}(\hat{C}_{BE})\ , \qquad (24)$$

where $\hat{C}_{SB}$ and $\hat{C}_{BE}$ are the CAVs of links *SB* and *BE*, respectively, and defines the left shift of $\hat{C}_{BE}$ by 2 d_{SB} (twice the propagation delay of link *SB* measured in τ_l-time units). Left shifting $\hat{C}_{BE}$ by d_{SB} positions purges utilization information corresponding to time periods that have already expired while left shifting it by another d_{SB} accounts for the propagation delay any burst sent from *S* suffers to reach node *B* (assuming the link propagation delay is the same in both directions). We finally execute a bit-wise AND operation, denoted by '**&**', between the *SB* and *BE* CAVs to compute the binary availability vector of the whole path *SBE*. This process is depicted in Figure 15.

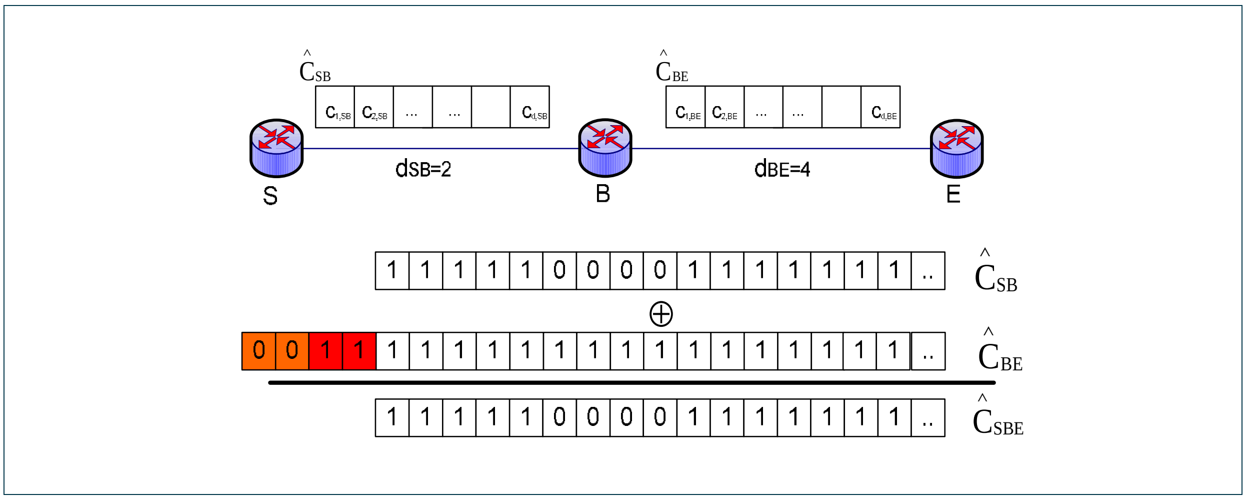

Figure 15: Calculation of the path capacity availability vector $\hat{C}_{SBE}$. $\hat{C}_{BE}$ is shifted by $2{\cdot}d_{SB}$ τ_l-time units (d_{SB}=2 in this example), before the AND operation is applied.

4.2.2 Binary Cluster Availability Vector over a Path

Let *p* be the path that starts at the distributed scheduler *S* and ends at the cluster *m*, and let $\hat{C}_p$ be its capacity availability vector and d_p be its delay. We want to transmit a task with data duration *b* ($b=I/C_l$ where *I* is the data size) over the path *p* in order to execute it at cluster *m*. We define $R_p(b)$ as the first position after which $\hat{C}_p$ has *b* consecutive ones. In other words, $R_p(b)$ is the earliest time after which a burst of duration *b* can start its transmission on path *p*. The earliest time that the task can reach cluster *m* is given by $\mathrm{EST}(p,b)=R_p(b)+b+d_p$. The distributed scheduler *S* has a partial (outdated) knowledge of the cluster availability vector $\hat{W}_m$ of *m*. We define $\mathrm{MUV}_k(\hat{W}_m)$ as the operation of setting zeros (making unavailable) the first *k* elements of vector $\hat{W}_m$. Then vector $\hat{W}_m(p,b)=\mathrm{MUV}_{\mathrm{EST}(p,b)}(\hat{W}_m)$ gives the time periods that *S* can schedule the task over path *p* at cluster *m*.

With respect to Figure 14 and Figure 15, we assume that we want to transmit a task of duration *b*=3 from *S* to the cluster at node *E*, over path p_{SBE} with propagation delay d_{SBE}=6. The capacity availability vector $\hat{C}_{SBE}$ was calculated in Section 4.2.1, and we have also calculated $R_{p_{SBE}}(b)=0$. The task reaches *E* after $\mathrm{EST}(p_{SBE},b)=R_{p_{SBE}}(b)+b+d_{SBE}=9$. Also, *S* has a (possibly outdated) knowledge of the cluster availability profile $\hat{W}_E$. The cluster availability vector that gives the periods that *S* can schedule the task on *E* is $\hat{W}_E(p_{SBE},b)=\mathrm{MUV}_{\mathrm{EST}(p_{SBE},b)}(\hat{W}_E)=\ \mathrm{MUV}_9(\hat{W}_E)$, which is the operation of setting the 9 first entries of vector $\hat{W}_E$ to zero. This process is depicted in Figure 16.

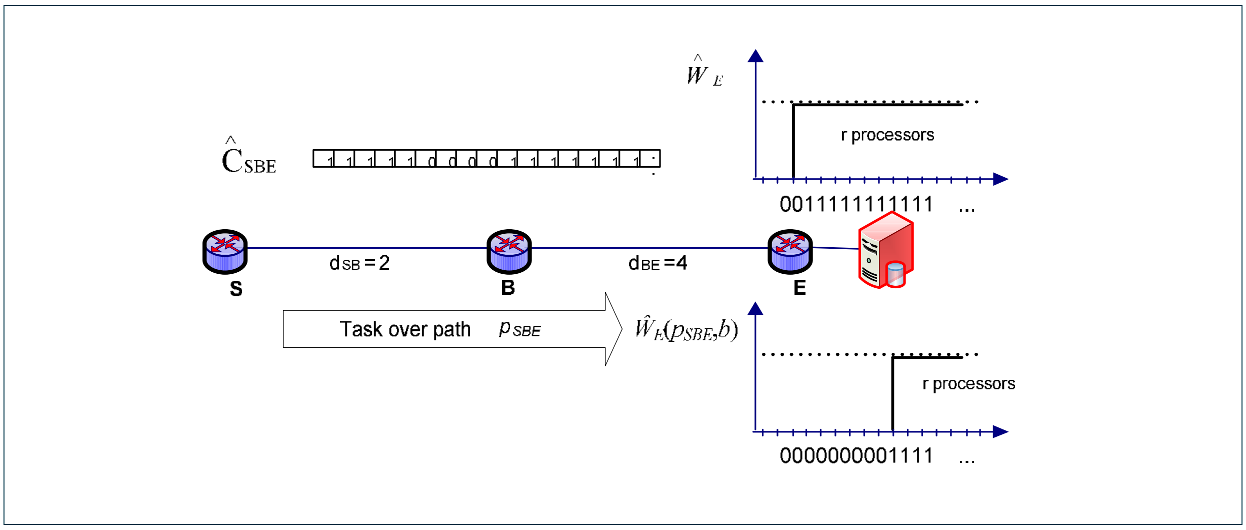

Figure 16: Scheduler S wants to transfer a task of data duration b=3 over path p_{SBE}. We denote by EST(p_{SBE},b) the earliest time the task can reach E over p_{SBE} and by $\hat{W}_E(p_{SBE},b)$ the cluster availability vector that gives the periods at which S can schedule the task on E. To calculate $\hat{W}_E(p_{SBE},b)$, we put 0's in the first EST(p_{SBE},b)=9 elements of $\hat{W}_E$.

4.3 Joint Communication and Computation Task Scheduling Algorithm in Grids

In what follows we present a multicost algorithm for the joint communication and computation scheduling of tasks. The algorithm consists of three phases. We first calculate the set $P_{n\text{-}d}$ of non-dominated paths between the source (which can be the scheduler S or a data repository site R) and all network nodes (Section 4.3.1). We then obtain the set $PM_{n\text{-}d}$ of candidate non-dominated (path, cluster) pairs from source to all the clusters that can process the task (Section 4.3.2). We finally choose from the $PM_{n\text{-}d}$ set the pair that minimizes the completion of the task execution, or some other performance criterion (Section 4.3.3).

4.3.1 Algorithm for Computing the Set of Non-Dominated Paths

In multicost routing, each link l is assigned a vector V_l of cost parameters, as opposed to the scalar cost parameter assigned in single-cost routing. In our initial formulation, the cost parameters of a link l include the propagation delay d_l of the link and its binary capacity availability vector $\hat{C}_l$, that is,

$$V_l = (d_l, \hat{C}_l) = (d_l, c_{1,l}, c_{2,l}, \ldots, c_{d,l}),$$

but they may also include other parameters (number of hops, number of tasks assigned to a cluster, etc). A cost vector [39] can then be defined for a path p consisting of links 1,2,…,k, based on the link cost vectors:

$$V(p) = \mathop{\mathbf{e}}_{l=1}^{k} V_l \overset{def}{=} \left(\sum_{l=1}^{k} d_l, \bigoplus_{l=1}^{k} \hat{C}_l \right), \tag{25}$$

where $\oplus$ is the associative operator defined in Eq. (24).

We say that path p_1 dominates path p_2 for a given burst and source-destination pair if the propagation delay of p_1 is smaller than that of p_2, and path p_1 is available for scheduling the burst (at least) at all time intervals at which path p_2 is available. Formally:

$$p_1 \text{ dominates } p_2 \ \ (\text{notation: } p_1 > p_2) \text{ iff } \sum_{l\in p_1} d_l < \sum_{l\in p_2} d_l \text{ and } \bigoplus_{l\in p_2} \hat{C}_l \leq \bigoplus_{l\in p_1} \hat{C}_l, \tag{26}$$

where the vector inequality "≤" should be interpreted component wise. The set of non-dominated paths $P_{n\text{-}d}$ for a given burst and source-destination pair is then defined as the set of paths with the property that no path in $P_{n\text{-}d}$ dominates another path in $P_{n\text{-}d}$.

An algorithm for obtaining the set $P_{n\text{-}d}$ of non-dominated paths from a given source (the scheduler *S* or a data repository site *R*) to all destination nodes is given in [39] and [38], and is a generalization of Dijkstra's algorithm that only considers scalar link costs.

4.3.2 Set of Non-Dominated (path, cluster) Pairs

In the first phase of our proposed routing and scheduling algorithm we obtain the set of non-dominated paths between the source (*S* or *R*) and all the nodes of the network. We now expand the definition of the path cost vector to include the utilization profiles of the clusters. More specifically, we define the cost vector of a (path, cluster) pair *pm* of a path *p* ending to a cluster *m* as:

$$V(pm) = \left(V(p),\ \hat{W}_m(p,b) \right) = \left(\sum_{l=1}^{k} d_l,\ \underset{l \in p}{\oplus} \hat{C}_l,\ \hat{W}_m(p,b) \right), \tag{27}$$

where $\hat{W}_m(p,b) = \mathrm{MUV}_{\mathrm{EST}(p,b)}(\hat{W}_m)$ is the binary cluster availability vector of *m* with 0's at the first EST(*p*,*b*) elements (Section 4.2.2).

We define a domination relationship between (path, cluster) pairs: A (path, cluster) pair p_1m_1 dominates another pair p_2m_2 for a given task, if p_1 dominates p_2, and also the cluster m_1 can schedule the task (after the minimum transmission delay over p_1) at least at all time intervals at which the cluster m_2 is available (after the minimum transmission delay over p_2). Formally:

$$p_1m_1 \text{ dominates } p_2m_2 \text{ (notation: } p_1m_1 > p_2m_2\text{) iff } p_1 > p_2 \text{ and } \hat{W}_{m_1}(p_1,b) \le \hat{W}_{m_2}(p_2,b), \tag{28}$$

where the vector inequality "≤" is interpreted component wise. The set of non-dominated (path, cluster) pairs $PM_{n\text{-}d}$ is then defined as the set of (path, cluster) pairs with the property that no pair in $PM_{n\text{-}d}$ dominates another. Clearly, we have $PM_{n\text{-}d} \subseteq P_{n\text{-}d}$. Therefore, to obtain the set $PM_{n\text{-}d}$ we apply Eq. (28) to the elements of $P_{n\text{-}d}$.

4.3.3 Finding the Optimal (path, cluster) Pair and the Transmission and Execution Time Offsets

In the third phase we apply an optimization function *f*(*V*(*pm*)) to the cost vector of each pair $pm \in PM_{n\text{-}d}$ to select the optimal path and cluster. The function *f* can be different for different tasks, depending on their QoS requirements. For example, if we want to the optimize data transmission, which corresponds to the routing optimization problem, the function *f* will select the path minimizing the reception time of the data at the cluster. If we consider the optimization of the computation problem, the function f will select the cluster that has the fewer scheduled tasks, or the one that minimizes its completion time. A combination of the above considerations can be also employed. Note that the optimization function *f* applied to a (path, cluster) cost vector to compute the final (scalar) cost has to be monotonic in each of the cost components. For example, it is natural to assume that it is increasing with respect to delay, decreasing with capacity, decreasing with increased capacity availability, decreasing with increased cluster availability, etc.

The next step is to choose from the set $PM_{n\text{-}d}$ of non-dominated *pm* pairs the one that minimizes *f*(*V*(*pm*)). In the context of this study we assume that we want to minimize the completion time of the task and that we are using a one-way connection establishment and reservation scheme. This is done in the following way:

Step I: Compute the first available position to schedule the task

We start from the cost vector $V(p_im_i)$ of pair p_im_i and calculate the first position $R_i(w_i)$ after which $\hat{W}_{m_i}(p_i,b)$ has $w_i = W/C_{mi}$ consecutive ones. In other words, $R_i(w_i)$ is the earliest time at which a task of computation workload W can start execution on m_i. Note that the way w_i is calculated accounts for the computation capacity of resource m_i, and that $\hat{W}_{m_i}(p_i,b)$, by definition, accounts for the earliest transmission time, the propagation delay of path p_i and the transmission delay (Section 4.2.2).

Step II: Select the cluster with the minimum task completion time

Select the pair $p_i m_i$ that results in the minimum completion time $R_i(w_i)+w_i$ for the task. In case of a tie, select the path with the smallest propagation delay. The time offset of task execution ($TO_{cluster}$) is given by $R_i(w_i)$.

Step III: Selecting the time to schedule the burst

Having chosen the pair $p_i m_i$ we transmit the task at the earliest time possible. The time offset TO_{path} for the data transmission is $R_{p_i}(b)$, defined as the first position after which $\hat{C}_{pi}$ has b consecutive ones (like in Section 4.2.2).

Step IV: Updating the CAV of chosen (path, cluster)

Having chosen the pm pair and the time offsets TO_{path} and $TO_{cluster}$ to transmit and execute the task, the next step is to update the utilization profiles of the corresponding links and the cluster. Update messages must also be sent to maintain the profiles at the other distributed schedulers. Such update mechanisms are extensively presented in [40] and [38] and are not described in this study.

The procedure described above assumes tell-and-go protocol. If we wish to use a tell-and-wait protocol we simply have to redefine $\hat{W}_{m_i}(p_i,b)$, $R_i(w_i)$, and $R_{p_i}(b)$ to take into account the round trip time before the data transmission.

4.4 Polynomial Algorithm for Computing the Set of Non-Pseudo-Dominated Paths

A serious drawback of the algorithm described in the previous section is that the number of non-dominated paths pairs may be exponential, and the algorithm is not guaranteed to finish in polynomial time. The basic idea to obtain polynomial time variations of this algorithm is to define a pseudo-domination relationship $>_{ps}$ between paths, which has weaker requirement than the domination relationship > defined in Eq. (28).

In [38] two such pseudo-domination relations were proposed and evaluated. For the scope of this study we present the better performing relation. We define a new link metric, called the slot availability weight of the link, as $weight(\hat{C}_l)$, which represents the total number of 1's in the vector $\hat{C}_l$.

The polynomial-time heuristic variation of the optimal multicost algorithm computes the set of non-pseudo-dominated paths following the same steps presented in Section 4.3.1. The algorithm still maintains the binary vectors of the paths but the domination relationship that is used to prune the paths is not Eq. (28) but the following:

$$p_1 \text{ pseudo-dominates } p_2\ (p_1 >_{ps} p_2) \text{ iff } \sum_{l\in p_1} d_l < \sum_{l\in p_2} d_l \text{ and } weight(\bigoplus_{l\in p_1} \hat{C}_l) > weight(\bigoplus_{l\in p_2} \hat{C}_l) \qquad (29)$$

When the domination relationship of Eq. (29) is used, an upper limit on the number of non-pseudo-dominated paths per source-destination pair is the dimension d_l of the capacity availability vectors. The heuristic algorithm obtained in this way, avoids the tedious comparisons of the CAVs of the optimal multicost algorithm, by essentially converting a d_l+1 dimensioned cost vector into a cost vector of dimension 2 that conveys most of the important information contained in the original vector. This problem was proven to be polynomial in [47].

4.5 Performance Results

In order to evaluate the performance of the proposed multicost algorithm for the joint communication and computation task scheduling and of its polynomial-time heuristic variation, we conducted simulation experiments, assuming an Optical Burst Switched network. We have extended the ns-2 platform [41] and tested the following routing algorithms:

- Optimal multicost algorithm for the joint communication and computation task scheduling (MC-T), as presented in Section 4.3.
- AW heuristic multicost algorithm for the joint communication and computation task scheduling (AWMC-T), as presented in Section 4.4.
- Optimal multicost burst routing and scheduling algorithm (MC-B), as presented in [38]. The MC-B algorithm takes into account only the communication part of the problem, and routes the input data to the cluster at which the data will arrive earlier, without using the related utilization profiles of the clusters.
- Earliest Completion time (ECT). The ECT algorithm considers only the computation part of the problem, and sends the task to the cluster where it will complete execution earlier, using the shortest path and examining contention only at the first link.

In order to establish the connection and reserve the appropriate communication and computation resources we have implemented a one-way reservation protocol capable of supporting advance reservations. The protocol is similar to JET [43] with extensions to cope with the one-way reservation of computation resources.

The simulations were performed assuming a 5x5 mesh network with wraparounds, where the nodes were arranged along a two-dimensional topology, with neighboring nodes placed at a distance of 400 km. In this topology we placed 4 clusters. Each cluster had 25 CPUs and each CPU had a computational capacity C_m = 25000 MIPS (typical value for Intel Xeon CPUs). We placed the clusters at nodes with coordinates (2,2), (2,4), (4,2), (4,4). Each link had a single wavelength of bandwidth C_l equal to 1 Gb/s. Users were placed at all the 25 nodes of the network and the tasks were generated according to a Poisson process with rate $\lambda/25$ tasks per second at each. The computation workload of each task was exponentially distributed with average value W M.I. (so the average task execution time is $w=W/C_m$). The size of the input data burst was also exponentially distributed with average I Bytes (so the average burst duration is $b=I/C_l$). The source of the input data I was always the scheduler. Finally, the used time-discretization steps and utilization vector dimensions (Section 4.1) for the links and clusters were: τ_l=0.001 sec, d_l=10000, and τ_m=0.04 sec, d_m=10000, respectively.

To assess the performance of the algorithms we used the following metrics:

- Average total delay: defined as the time between the task creation and its execution completion time.
- Burst blocking probability: the probability of an input data burst to content with another burst.
- Conflict probability: the probability a task finds a cluster unavailable at the time predicted by the algorithm (equal to $TO_{cluster}$ - Section 4.3.3), due to another task that has already reserved that cluster (the so called race problem).

Note that if we used a centralized architecture the burst blocking and conflict probabilities would be negligible, since the "central" scheduler would have a complete knowledge of the utilization of the resources and would schedule the tasks accordingly. Thus, these two metrics depend on the employed update strategy, as well as the examined algorithm.

We used the following parameters: b = 1 sec and w=10 sec. We classify the tasks as CPU- and data-intensive since w is considerable with respect to the total computation power, while b is considerable with respect to the total communication capacity of the network. The average total delay can take values less than 11sec, since a task of a user that is attached to a node which also is a cluster can be executed locally (without data transfers).

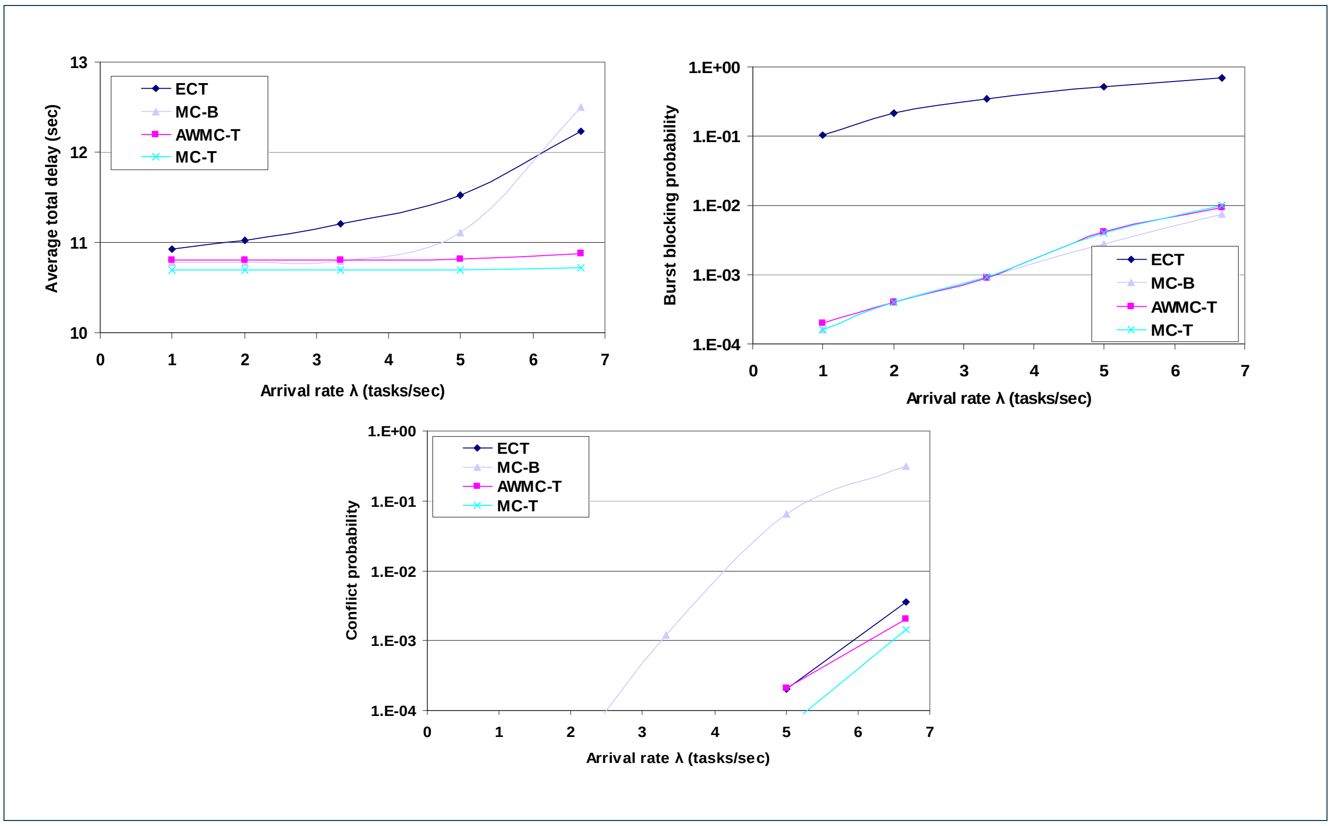

Figure 17: Performance for tasks that are CPU- and data-intensive: (a) average total delay, (b) burst blocking probability and (c) conflict probability.

In Figure 17.a we observe that the multicost algorithms that jointly consider the communication and computation resources (MC-T, AWMC-T) perform better than the other two algorithms (MC-B, ECT) with respect to the average total delay metric. The average total delay of the MC-T and AWMC-T algorithms increase slightly with the tasks' generate rate *λ*. The tasks have high demands for both communication and computation resources and these algorithms solve this joint problem efficiently as can be seen by the corresponding low burst blocking probability (Figure 17.b) and the low conflict probability (Figure 17.c). On the other hand, the performance of ECT deteriorates as *λ* increases. ECT does not take into account the communication part of the problem, and thus exhibits a high burst blocking probability as *λ* increases (Figure 17.b) which results in increased average total delay. Similarly, the performance of the MC-B algorithm deteriorates as λ increases. MC-B does not take into account cluster availability, and the clusters chosen are usually not the optimum ones, as can be seen by the high conflict probability (Figure 17.c), which introduces additional delay to the total time of the task.

From these results it is clear that in a Grid network where tasks are both CPU- and data-intensive (or where some tasks are CPU-intensive and some data-intensive) performance improves significantly by jointly optimizing the use of the communication and computation resources.

From Figure 17 we can conclude that the difference in the average delay performance between the optimal multicost (MC-T) and the heuristic multicost (AWMC-T) algorithms is small. In Figure 18.a we graph the average number of searched paths per task request (that is, the average size of the set P_{n-d}, presented in Section 4.3.1). We observe that the optimal multicost algorithm searches significantly more paths than the heuristic algorithm and the difference increases as the load (expressed by *λ*) increases. Figure 18.b shows the average number of operations required to route and schedule a task by the two algorithms. An operation is defined as an addition, a Boolean operation (e.g., AND) or a comparison ($<$, $>$, $\neq$, etc). Note that the MC-T, AWMC-T and MC-B algorithms use as cost parameters the delay and the link (or path) utilization profiles and thus can be viewed as multicost algorithms with $1+d_l$ cost parameters. Although reducing the value of d_l would result in better complexity, the algorithms' blocking performance would deteriorate since this would constrain their capability of advance scheduling. The value d_l=10000 was chosen so as to optimize the blocking probability. Specifically, in all the experiments that we conducted all task requests were able to find available (temporal) placements for their input data. As expected the heuristic algorithm requires fewer operations than the optimal multicost algorithm. Thus, the proposed polynomial time AW multicost algorithm yields delay performance that is very close to that of the optimal multicost algorithm, while maintaining the number of searched paths and required operations at low levels.

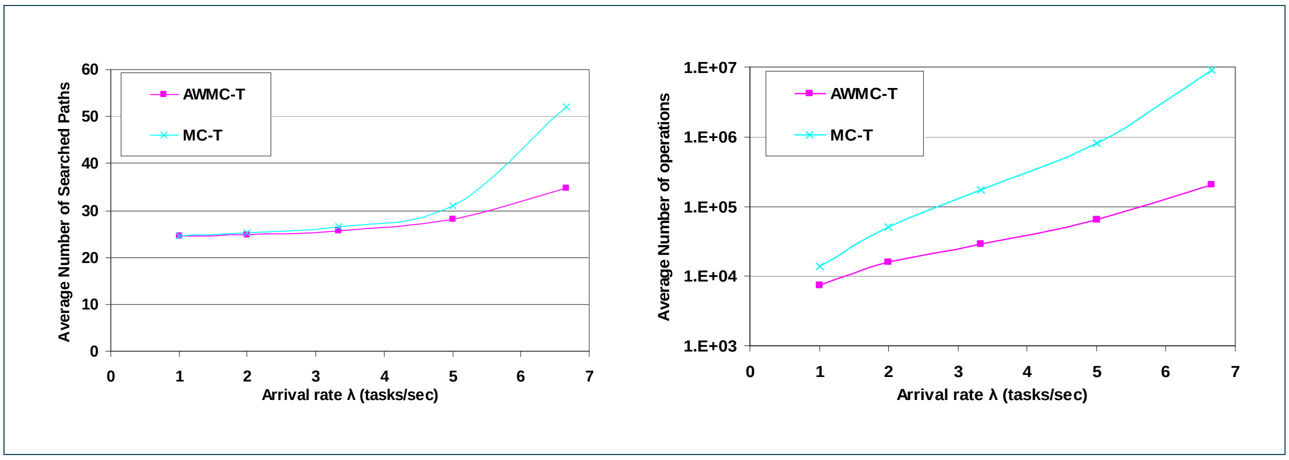

Figure 18: Algorithms complexity: (a) average number of searched paths, and (b) average number of operations.

5 Conclusion

We proposed and evaluated Quality of Service (QoS) - aware and fair scheduling algorithms for Grids, which are capable of optimally mapping tasks to resources, considering the task characteristics and QoS requirements. We mainly addressed scheduling problems on computation resources, but also looked at the joint scheduling of communication and computation resources.

Our proposed algorithms were designed to serve both tasks with hard QoS requirements (referred to as Guaranteed Service - GS tasks) and tasks with no QoS requirements (referred to as Best Effort - BE tasks). In Section 2 we proposed scheduling algorithms that either offer a fair degradation in the QoS GS tasks are receiving in case of congestion, or allocate resources to BE tasks in a fair way. The notion of fairness we used was that of Max-Min sharing. Fairness was provided either on a per user basis or on a per task basis. Task demands (such as deadlines) were also incorporated in our definition of fairness. An extensive number of simulations were performed to compare the proposed algorithms with traditional scheduling schemes. The results indicate that our proposed algorithms provide fairness, while taking the QoS requirements of the users into account and better exploiting the available resources.

In Section 3 we presented a framework for providing hard (deterministic) delay guarantees to GS users. The GS users are leaky bucket constrained, so as to follow a (ρ, σ) constrained task generation pattern, which is agreed separately with each resource during a registration phase. The delay guarantees imply that a GS user can choose a resource to execute his task before its deadline expires, with absolute certainty. We also proposed and evaluated schemes for the categorization of computational resources that serve either GS, or BE, or both types of users, with varying priorities. Our simulation study, which used data from a real Grid Network, indicates that our framework succeeds in providing hard delay guarantees to the GS users as long as they respect their constraints, even with small deviations. We examined several resource allocation scenarios and found that the use of resources that serve both GS and BE users with varying priorities, results in fewer missed deadlines and better resource usage. Scheduling without a-priori knowledge of task workloads was also examined.

The joint scheduling of communication, computation, storage and other resources is another very important topic in Grids. To address the problem of successive scheduling of communication and computation resources we presented in Section 4 a multicost algorithm that uses advance reservations on the corresponding resources. We initially presented an optimal scheme of non-polynomial complexity and by appropriately pruning the set of candidate paths we also obtained a heuristic algorithm of polynomial complexity. We showed that in a Grid network where the tasks are CPU- and data-intensive important performance benefits can be obtained by jointly optimizing the use of the communication and computation resources as our proposed algorithms do. The proposed heuristic algorithm was shown to combine the strength of the optimal multicost algorithm with a low computation complexity.

6 References

[1] I. Foster and C. Kesselman, “The Grid: Blueprint for a New Computing Infrastructure”, 2nd Edition, Morgan Kaufman, 2003.

[2] S. Zhuk, A. Chernykh, A. Avetisyan, S. Gaissaryan, D. Grushin, N. Kuzjurin, A. Pospelov, and A. Shokurov, ”Comparison of Scheduling Heuristics for Grid Resource Broker”, In Proceedings of the Fifth Mexican international Conference in Computer Science (Enc’04), Washington, DC, pp. 388-392.

[3] R. Buyya, D. Abramson, J. Giddy, and H. Stockinger, "Economic Models for Resource Management and Scheduling in Grid Computing," Special Issue on Grid Computing Environments, The Journal of Concurrency and Computation: Practice and Experience (CCPE), Vol. 14, pp. 1507-1542, 2002.

[4] R. Buyya, D. Abramson, and S. Venugopal, “The grid economy,” In Special Issue on Grid Computing; Proceedings of the IEEE, Vol. 93, No. 3, pp. 698–714.

[5] R. Yahyapour, P. Wieder, “Grid Scheduling Use Cases”, Grid Scheduling Architecture Research Group (GSA-RG), Open Grid Forum (OGF), March, 2006.

[6] J. Xiao, Y. Zhu, L. M. Ni, and Z. Xu, “GridIS: An incentive-based Grid scheduling”, In Proceedings of the 19th International Parallel and Distributed Processing Symposium (IPDPS’05), 2005.

[7] V. Subramani, R. Kettimuthu, S. Srinivasan and P. Sadayappan, “Distributed job scheduling on computational Grids using multiple simultaneous requests”, In Proceedings of the 11th IEEE Symposium on High Performance Distributed Computing, 2002.

[8] Y. Cardinale, H. Casanova, “An evaluation of Job Scheduling Strategies for Divisible Loads on Grid Platforms”, In Proceedings of the High Performance Computing & Simulation Conference (HPC&S’06), Bonn, 2006.

[9] R. Braden, D. Clark, and S. Shenker, “Integrated services in the internet architecture: an overview”, 1994 RFC 1633, Internet Engineering Task Force.

[10] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang and W. Weiss, “An Architecture for Differentiated Service”, 1998, RFC 2475, Internet Engineering Task Force.

[11] I. Foster, C. Kesselman, C. Lee, R. Lindell, K. Nahrstedt and A. Roy, “A Distributed Resource Management Architecture that Supports Advance Reservation and Co-Allocation”, In Proceedings of IWQOS, UK, 1999, pp. 27-36.

[12] S. N. Bhatti, S.-A. Sorensen, P. Clarke, J. Crowcroft, “Network QoS for Grid Systems”, International Journal of High Performance Computing Applications, Special Issue on "Grid Computing: Infrastructure and Applications.", 2003, Vol. 17, No. 3, pp 219-236.

[13] H. Chu, “CPU Service Classes: a Soft Real Time Framework for Multimedia Applications”, University of Illinois at Urbana-Champaign, Technical Report, 1999.

[14] R. Ali, O. Rana, D. Walker, S. Jha and S. Sohail, “G- QoSM: Grid Service Discovery using QoS Properties”, Journal of Computing and Informatics, Special issue on "Grid Computing", 2002, Vol. 21, No. 4, pp. 363-382.

[15] A.K Parekh, R.G Gallager, "A generalized processor sharing approach to flow control in integrated services networks: the single-node case," IEEE/ACM Transactions on Networking, 1993, Vol. 1, No. 3, pp. 344 -357.

[16] A. Demers, S. Keshav and S. Shenker, “Design and Analysis of a Fair Queuing Algorithm,” In Proceedings of the ACM SIGCOMM, Austin, 1989.

[17] D. Bertsekas, R. Gallager, Data Networks, 2nd edition, Prentice Hall 1992 (section starting on p.524).

[18] K. Rzadca, D. Trystram, A. Wierzbicki, “Fair Game-Theoretic Resource Management in Dedicated Grids”, International Symposium on Cluster Computing and the Grid, 2007, pp. 343-350.

[19] K. H. Kim, R. Buyya, “Fair Resource Sharing in Hierarchical Virtual Organizations for Global Grids”, In Proceedings of the 8th IEEE/ACM International Conference on Grid Computing, 2007.

[20] L. Amar, A. Barak, E. Levy, M. Okun, “An On-line Algorithm for Fair-Share Node Allocations in a Cluster”, In Proceeding of International Symposium on Cluster Computing and the Grid, 2007, pp. 83-91.

[21] N. Doulamis, A. Doulamis, A. Panagakis, K. Dolkas, T. Varvarigou and E. Varvarigos, “A Combined Fuzzy -Neural Network Model for Non-Linear Prediction of 3D Rendering Workload in Grid Computing,” IEEE Transactions on Systems, Man and Cybernetics, 2004, Part-B, Vol. 34, No. 2, pp. 1235-1247.

[22] N. Doulamis, A. Doulamis, E. Varvarigos, and T. Varvarigou, “Fair QoS Resource Management in Grids”, to appear in IEEE Transactions on Parallel and Distributed Systems.

[23] B. Briscoe, “Flow Rate Fairness: Dismantling a Religion”, In Proceeding of SIGCOMM Computer Communication Review, 2007, Vol. 37, No. 2, pp. 63-74.

[24] K. Christodoulopoulos, V. Gkamas, E. Varvarigos, “Statistical Analysis and Modeling of Jobs in a Grid Environment”, to appear in Grid Computing.

[25] R. Buyya, M. Murshed, “GridSim: A Toolkit for the Modeling and Simulation of Distributed Resource Management and Scheduling for Grid Computing”, Concurrency and Computation: Practice and Experience (CCPE), 2002, Vol. 14, No. 13-15, pp. 1175-1220.

[26] R. Raman, M. Livny, and M. Solomon, “Policy driven heterogeneous resource co-allocation with gangmatching,” In Proceedings of the 12th IEEE International Symposium on High Performance Distributed Computing (HPDC-12), 2000, pp. 80–89.
[27] K. Czajkowski, I. T. Foster, and C. Kesselman, “Resource co-allocation in computational grids,” In Proceedings of 8th IEEE International Symposium on High Performance Distributed Computing (HPDC-8), 1999, pp. 219-228.
[28] A. Andrieux, K. Czajkowski, A. Dan, K. Keahey, H. Ludwig, T. Nakata, J. Pruyne, J. Rofrano, S. Tuecke, and M. Xu, “Web Services Agreement Specification (WS-Agreement)”, 2007, https://forge.Gridforum.org/ sf/docman/do/downloadDocument/projects.graapwg/docman.root.current drafts/doc6091.
[29] J. Yu and R. Buyya, “A Taxonomy of Scientific Workflow Systems for Grid Computing”, Special Issue on Scientific Workflows, SIGMOD Record, 2005, Vol. 34, No. 3, pp. 44-49,.
[30] T. Tannenbaum, D. Wright, K. Miller, and M. Livny, “Condor – A Distributed Job Scheduler”, Beowulf Cluster Computing with Linux, The MIT Press, MA, USA, 2002.
[31] J. Cao, S. A. Jarvis, S. Saini, G. R. Nudd, “GridFlow: Workflow Management for Grid Computing”, In Proceedings of 3rd International Symposium on Cluster Computing and the Grid (CCGrid), Tokyo, Japan, 2003.
[32] R. Buyya and S. Venugopal, “The Gridbus Toolkit for Service Oriented Grid and Utility Computing: An Overview and Status Report”. In 1st IEEE International Workshop on Grid Economics and Business Models, GECON 2004, Seoul, Korea, pp. 19-36.
[33] K. Ranganathan, I. Foster. “Decoupling computation and data scheduling in distributed data-intensive applications”, HPDC, 2002.
[34] S. Venugopal, R. Buyya, L. Winton, “A Grid Service Broker for Scheduling Distributed Data-Oriented Applications on Global Grids”, MGC, 2004.
[35] K. Nahrstedt, H. Chu, S. Narayan, “QoS-aware resource management for distributed multimedia applications”, J. High Speed Networks, pp. 229-257.
[36] R. Guerin, A. Orda, “Networks with Advance Reservations: The Routing Perspective”, In Proceedings of the 19th Annual Joint Conference of the IEEE Computer and Communications Societies INFOCOM, 2000.
[37] Job Submission Description Language specification: www.ogf.org/documents/GFD.56.pdf.
[38] E. Varvarigos, V. Sourlas, K. Christodoulopoulos, “Routing and Scheduling Connections with Advance Reservations”, submitted to Computer Networks.
[39] F. Gutierrez, E. Varvarigos, S. Vassiliadis, "Multicost Routing in Max-Min Fair Networks", Allerton Conference, 2000.
[40] K. Manousakis, V. Sourlas, K. Christodoulopoulos, E. Varvarigos, K. Vlachos, “A Bandwidth monitoring mechanism: Enhancing SNMP to record Timed Resource Reservations”, Journal of Network and Systems Management, 2006, pp. 583-597.
[41] The Network Simulator (ns2): www.isi.udu/nsnam/ns.
[42] J. Turner, “Terabit burst switching,” Journal of High Speed Networks, 1999, Vol. 8, No. 1, pp. 3-16.
[43] C. Qiao, M. Yoo, “Optical burst switching (OBS)–a new paradigm for an optical Internet,” Journal of High Speed Networks,1999, Vol. 8, No. 1, pp. 69–84.
[44] Grid Optical Burst Switched Networks: www.ogf.org/Public_Comment_Docs/Documents/Jan-2007/OGF_GHPN_GOBS_final.pdf
[45] E. Varvarigos, V. Sharma, “An efficient reservation connection control protocol for gigabit networks”, Journal of Computer Networks and ISDN Systems, 1998, pp. 1135–1156.
[46] P. van Mieghem, and F. Kuipers, “Concepts of Exact QoS Routing Algorithms”, IEEE/ACM Transactions on Networking, 2004, Vol. 12, No. 5, pp. 851-864.
[47] Z. Wang, J. Crowcroft, “Quality-of-service routing for supporting multi-media applications”, Journal on Selected Areas in Communications, 1996, Vol. 14.
[48] L. Burchard, “Analysis of Data Structures for Admission Control of Advance Reservation Requests”, IEEE Transactions on Knowledge and Data Engineering, 2005, Vol. 17, No. 3, pp. 413–424.
[49] C. Barz, T. Eickermann, M. Pilz, O. Wäldrich, L. Westphal, W. Ziegler, “Co-Allocating Compute and Network Resources-Bandwidth on Demand in the VIOLA Testbed”, CoreGRID Symposium, Springer, 2007, CoreGRID Series, Towards Next Generation Grids, pp. 193–202.
[50] VIOLA – Vertically Integrated Optical Testbed for Large Application in DFN, 2007, http://www.viola-testbed.de/
[51] A. Streit, D. Erwin, Th. Lippert, D. Mallmann, R. Menday, M. Rambadt, M. Riedel, M. Romberg, B. Schuller, and Ph. Wieder, “UNICORE - From Project Results to Production Grids”, Grid Computing: The New Frontiers of High Performance Processing, Advances in Parallel Computing 14, Elsevier, 2005.
[52] P. Kokkinos, E. Varvarigos, “Resources Configurations for providing Quality of Service in Grid Computing”, CoreGRID Workshop on Grid Programming Model, Grid and P2P Systems Architecture, Grid Systems, Tools and Environments, Heraklion - Crete, Greece, 2007.
[53] P. Kokkinos, E. Varvarigos, N. Doulamis, “A Framework for Providing Hard Delay Guarantees in Grid Computing”, accepted in e-Science 2007.
[54] P. Kokkinos, E. Varvarigos, “A Framework for Providing Hard Delay Guarantees and User Fairness in Grid Computing”, submitted in IEEE Transactions on Parallel and Distributed Systems.
[55] H. Dafouli, P. Kokkinos, E. Varvarigos, “Fair Completion Time Estimation Scheduling in Grid Networks”, submitted to CCGrid 2008.
[56] K. Christodoulopoulos, N. Doulamis, E. Varvarigos, “Joint Communication and Computation Task Scheduling in Grids”, submitted to CCGrid 2008.