034115

PHOSPHORUS

Lambda User Controlled Infrastructure for European Research

Integrated Project

Strategic objective:
Research Networking Testbeds

Deliverable reference number D.5.2

QoS-aware Resource Scheduling

Due date of deliverable: 2007-09-30
Actual submission date: 2007-09-30
Document code: Phosphorus-WP5-D.5.2

Start date of project:
October 1, 2006

Duration:
30 Months

Organization name of lead contractor for this deliverable:
Research Academic Computer Technology Institute (CTI)

Project co-funded by the European Commission within the Sixth Framework Programme (2002-2006)		
Dissemination Level		
PU	Public	PU
PP	Restricted to other programme participants (including the Commission	
RE	Restricted to a group specified by the consortium (including the	
CO	Confidential, only for members of the consortium (including the Commission Services)	

Abstract

Grids offer a transparent interface to geographically scattered communication, computation, storage and other resources. The Phosphorus network concept and test-bed make users and applications aware of their complete (computational and networking) Grid resources, environment and capabilities, enabling dynamic, adaptive and optimized use of heterogeneous network infrastructures connecting various high-end resources**.** The Phosphorus project focuses on applications that have heavy communication requirements. However, many of these applications also have significant computational requirements. Working Package 5 – WP5 (Supporting Studies) performs research and studies to support the experimental activities of the Phosphorus project. Specifically routing, resource management, various advance reservation techniques, and scheduling algorithms are proposed and evaluated. Also recommendations for the design of an optical Grid control plane are provided.

In this deliverable we propose and evaluate QoS-aware and fair scheduling algorithms for Grid Networks. These algorithms are capable of optimally or near-optimally assigning tasks to resources, taking into account the task characteristics and QoS requirements. We categorize Grid tasks (also Grid users or applications) 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 requirements are referred to as Best Effort (BE) tasks. The algorithms we propose are designed to serve both kinds of tasks. Specifically, we propose scheduling algorithms that try to meet task deadlines, or offer fair degradation in the QoS GS tasks receive in case of congestion, or allocate resources to BE tasks in a fair way. We also propose a QoS-aware framework for Grid Networks that provides hard delay guarantees to GS tasks. Though, we mainly address scheduling problems on computation resources, we also look at joint scheduling of communication and computation resources. The corresponding routing and scheduling algorithms aim at satisfying of two or more QoS requirements, by co-allocating resources either concurrently or successively taking into account dependencies between communication and computation tasks.

List of Contributors

Emmanouel Varvarigos	CTI
Panagiotis Kokkinos	CTI
Konstantinos Christodoulopoulos	CTI
Markus Pilz	UBN
Christoph Barz	UBN
Tim Stevens	IBBT
Joachim Vermeir	IBBT
Chris Develder	IBBT
Marc De Leenheer	IBBT
Bart Dhoedt	IBBT

Table of Contents

Table of Figures

0 Executive Summary

The emergence of high speed optical networks is making the vision of Grids a reality. Grids offer a transparent interface to geographically scattered computational and storage resources. Resource scheduling is a key to the success of Grid networks, since it determines the efficiency with which resources are used and the Quality of Service (QoS) provided to the users. Today's Grids 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 applications and complex scientific applications with strict delay, computational power, and other requirements. Best effort service also limits the economic importance of Grids, since users will be reluctant to pay, directly or indirectly (e.g., by contributing resources to the Grid) for the service they receive, if they are not given performance guarantees. As a result, there is a growing need for Grid scheduling and resource management algorithms to be able to provide QoS to the users. In this deliverable we propose and evaluate various scheduling algorithms capable of optimally mapping tasks[1] to resources, considering the task characteristics and the QoS requirements of the users.

In Section 1 the motivation and the objectives of this deliverable are stated. We categorize the scheduling algorithms according to various criteria and report on previous work in the field. We present the user QoS requirements in a Grid environment and the scheduling usage patterns as identified by Open Grid Forum (OGF). We classify the algorithms to be presented in this deliverable with respect to the previously mentioned categorizations. Finally, we describe our contributions and the way they relate to other work carried out within the Phosphorus project.

Section 2 deals with fair scheduling policies. Most of the scheduling algorithms proposed to date try to minimize the average total task delay, maximize resources utilization, or maximize schedulability, without taking into account fairness considerations. Since the sharing of resources is the "raison d' etre" of Grids, fairness is a concept that is inherent in Grid scheduling, but has long been previously ignored. The scheduling algorithms proposed in Section 2 are centralized and consist of two-phases, the "task-ordering" phase and the "task-to-resource assignment" phase. In Section 2.1, two new fair scheduling algorithms for the first scheduling phase are proposed and evaluated through simulations. These algorithms define fairness based on the Max-Min fair sharing concept. In Section 2.2 we describe another scheduling algorithm, called Fair Completion Time Estimation Algorithm (FCTE), which applies fairness in the second phase of the two-phase scheduling

[1] In this document the terms "task" and "job" are used interchangeably.

procedure. Finally, in Section 2.3 we propose a scheduling algorithm for the first scheduling phase that provides fairness on a per user basis, instead of on a per task basis.

In Section 3 we import ideas and concepts from Data to Grid Networks and propose a QoS scheduling framework for two kinds of Grid users. We distinguish the users in Guaranteed Service (GS) and Best Effort (BE) users. We also categorize resources according to the type of users they serve (GS or BE or both) and the priority they give to each type. Our framework aims at providing deterministic service guarantees to GS users and fairness to BE 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. On the resources, the arriving GS tasks are first queued in a Weighted Fair Queuing (WFQ) scheduler in such a way that guaranteed task service rates can be given to each GS user in the same way WFQ provides guaranteed bandwidth in Data Networks. BE users, on the other hand, are handled by our framework with fairness as the main goal.

In Section 4 we address the problem of co-allocating resources (more than one and possible of different kind) in a Grid environment, either concurrently or with various interdependencies. We formulate different problems of co-allocation and propose optimal or near-optimal solutions to these problems.

In Section 4.1 we address the “Anycast Communication and Computation” problem, defined as follows: Given the Grid network we want to find the computation resource to execute a task and the path over which to route the data of the task so as to minimize some performance criteria. We present two variations to this problem and two algorithms to address these variations. The first is a multi-constrained routing and scheduling algorithm, which uses the path delay and the computation load as selection metrics. The network and the computation resources are concurrently co-allocated (allocated in the same time-frame). The second algorithm is a multicost algorithm to be used in the case that task processing is decomposed in two successive phases: (i) the transfer of data from the scheduler or a data repository site to the computation resource and (ii) the execution of the task. The proposed algorithm selects the computation resource to execute the task, determines the path to route the input data, finds the starting times for the data transmission and the task’s execution, and performs advance reservations on the corresponding communication and computation resources.

In Section 4.2 we address a problem of concurrent co-allocation of network, storage and computation resources. More specifically, each task is characterized by its processing rate, the size of the input (output) datasets, and the input (output) bandwidth. Scheduling is performed at periodic instants. We present an ILP formulation that performs two functions: task admission control and resource assignment. Thus, a solution of the ILP model tells us whether or not a task is accepted for execution, and if so, which resources it may use. Accepted tasks are all started immediately; rejected tasks are transferred automatically to the next scheduling round.

In Section 4.3 we present the MetaScheduling Service (MSS) developed in VIOLA, which allows the end-user to execute the individual components of his application using the most appropriate resources available. The orchestration of resources of different sites belonging to different administrative domains is done by the MSS. This service is responsible for the negotiation of agreements on resource usage with the individual local resource management systems, using WS-Agreement. The MSS developed in VIOLA will be enhanced to cope with the requirements of the applications in the Phosphorus project. We turn our attention to the advance reservation of network resources and present a brief overview of the network reservation system ARGON developed in the VIOLA project. ARGON includes an advance reservation capable interface for the Grid application layer and thus offers connectivity services with a specified QoS on top of the optical network.

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 deliverable we propose and evaluate QoS-aware and fair scheduling algorithms for 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 Virtual Organizations (VOs), and request the services of the Grid for their execution. Tasks may depend or not on each other and may require the use of different kind of resources, such as, computation, network, or storage resources, or specific instruments. The efficiency of a task's execution is defined in various ways in Section 1.3.1.

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, abbreviated RB) selects the resources that a task will use. These can be computational, communication, storage or other resources. At the second level, which is usually called local scheduling, each resource (or, 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 deliverable we are more interested in the first level of Grid scheduling, namely meta-scheduling, which is also the most challenging to design and optimize.

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 assignment. In order to assign the tasks efficiently, the central meta-scheduler may use information on the utilization of the resources and the tasks requirements. In a distributed scheme, there is no central meta-scheduler, but every user site has a meta-scheduler (which we call distributed meta-scheduler), and the assignment decision is taken 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 scheduling 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 centralized, distributed and hierarchical scheduling architectures.

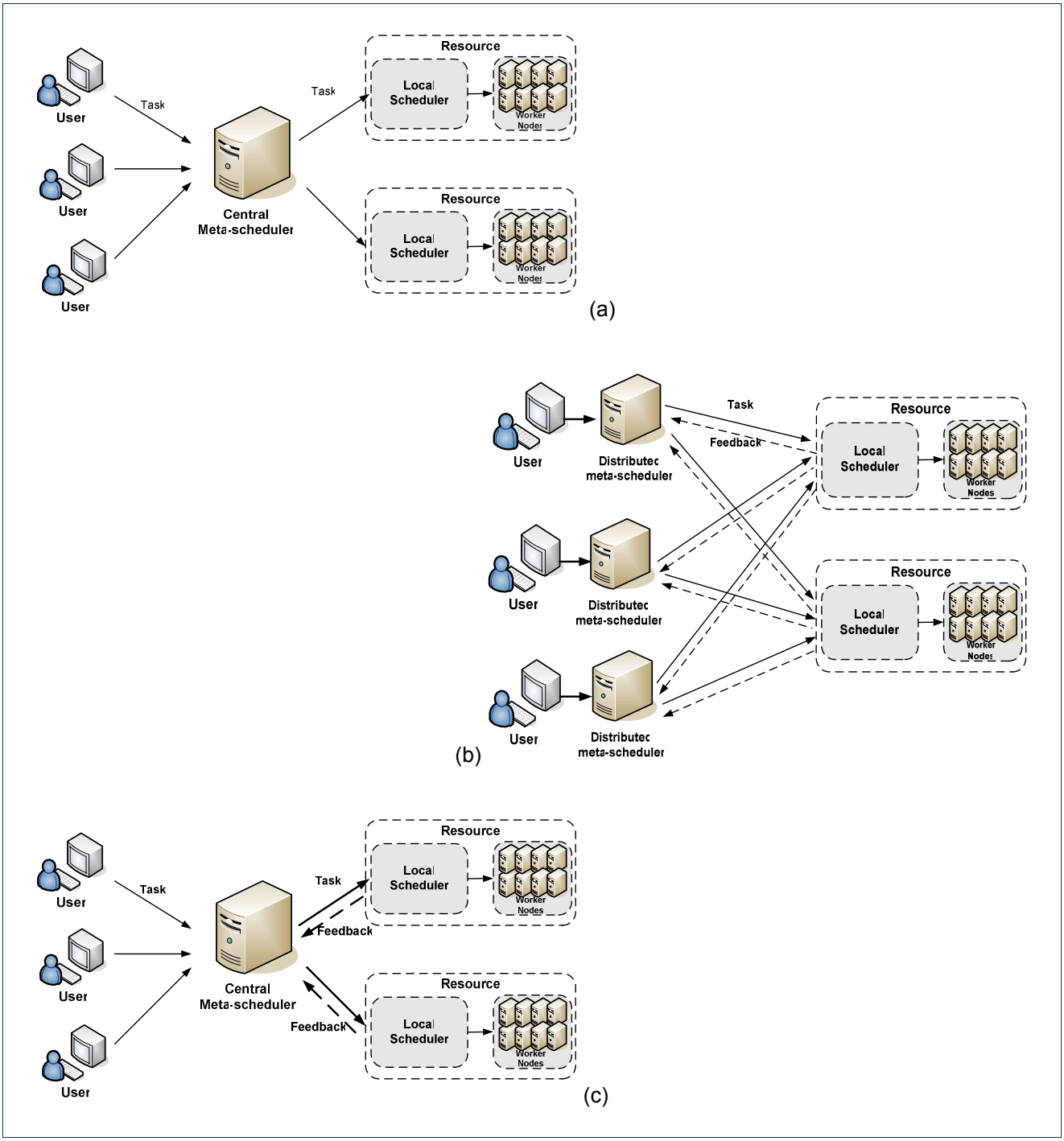


Figure 1 - (a) Centralized (b) Distributed and (c) Hierarchical Grid scheduling architectures

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 so that several tasks are accumulated 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).

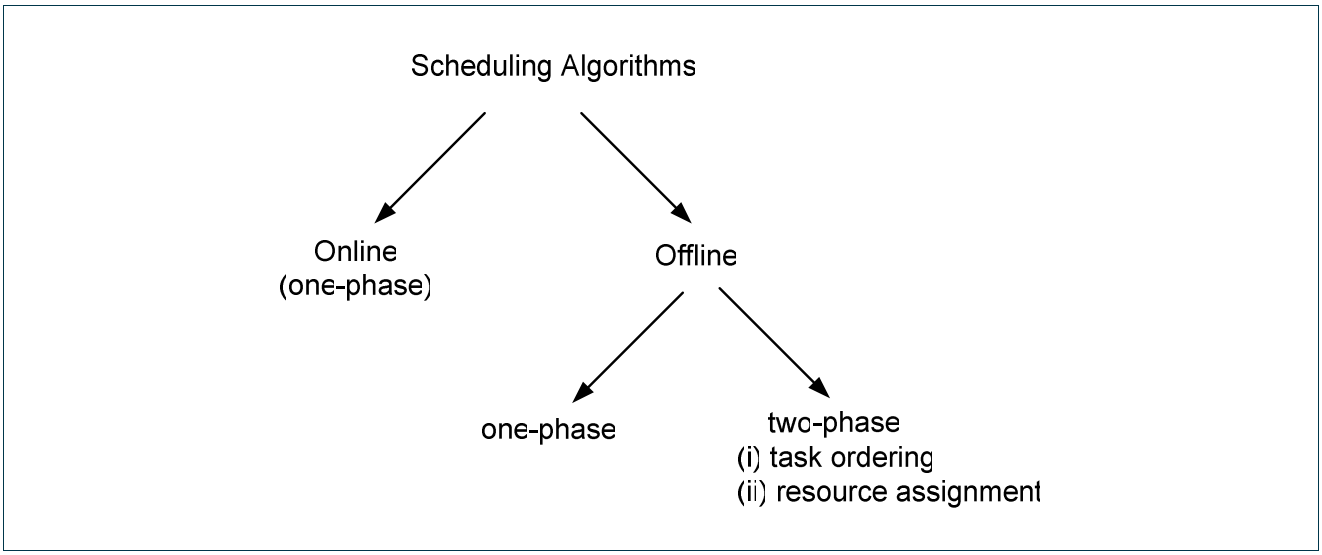


Figure 2 – Categorization of Grid scheduling algorithms into “online” and “offline” algorithms.

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 deliverable we mainly address scheduling problems on computation resources, we also look at joint scheduling of communication and computation resources.

1.3 Grid User Requirements and Scheduling Usage Patterns

In order to design useful, efficient and practical Grid scheduling algorithms, we should first consider the Quality of Service 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 requirements include functional requirements, such as security, resource discovery, fault management, performance, event monitoring and others, in this deliverable 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, or for selecting a feasible path over which to route the task or its data, or for performing resources coordination and advance reservations, etc.

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 deliverable 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. The communication delay includes the delays incurred for the transfer of the task's input and output data to and from the computation resource. The computation delay is the time it takes for the execution of the task in a computation resource including the queuing time and the execution (processing) time. In the case of a more complicated application, additional delays exist. For example, a task may wait for the intermediate results produced by the execution of other related tasks. Usually the user describes his 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 components may also be requested.

- **Delay Jitter:** Delay jitter is the variation of the total delays of the submitted tasks. In packet switched networks, continuous-media (video, audio, image) streams require for good quality of reception that jitter will be kept below a sufficiently small upper bound. 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 period of time the user wants to. 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, either due to the large number of users that are already served or other issues, then the task cannot be scheduled in the Grid. Schedulability, defined as one minus the rejection probability, is also a term 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 capacity requirement of a user depends on how 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 such an environment the user will not choose a specific 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. A Grid user forwards his requirements along with a description of some task characteristics to a meta-scheduler, which processes them in order to find if their satisfaction is possible. This process includes a number of algorithms for selecting a suitable computation site for the execution of the task, for selecting a feasible path over which to route the task and of course algorithms for resources coordination and possibly for in-advance reservations.

Note that fairness is an inherent requirement for both GS and BE users. The GS user should receive a graceful and fair degradation in their QoS they experience, if there are limited computation and communication resources. Moreover, it is desirable for the Grid Network to allocate resources to BE users 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 deliverable follow to a great extent the Grid scheduling usage patterns that are presented here. Thus, in each of the following three chapters of this deliverable 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 - VO) submits a task, on the Grid Network, which has no dependencies but requires a specific service guarantee. Such service guarantees can be the task's deadline, the number of CPUs the task needs for its execution, the storage size it needs in order to store its 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. Some of these mechanisms are: advance (or not) reservations, queuing mechanisms combined with a backfilling strategy, admission control and others. The most 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. one based on WS-Agreement), 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 deliverable, investigate the objectives that they try to satisfy and the way these algorithms can fit in the Phosphorus testbed.

In Section 2 we examine a number of fair scheduling algorithms. Fairness can be defined in a number of different 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. Of course, in cases that the Grid network serves different classes of users (e.g., users willing to pay different amounts of money) fairness can be proportional to the class of the user, while users belonging to the same class should have 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 the Grid users. Delay is one of the most common requirements the users impose in the execution of their tasks. The proposed framework concentrates more on the computational than on the communication resources. In order for the framework to provide hard delay guarantees, the 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 the framework provides hard 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, as described in Section 1.3.2.2. The proposed framework can be employed in a centralized or a distributed scheduling architecture and uses an online algorithm to assign the tasks to resources.

In Section 4 we address the problem of co-allocating resources in a Grid environment, either in advance or concurrently. Thus the algorithms presented in this section fall in the "Task requesting many service guarantees" usage pattern described in Section 1.3.2.3.

In Section 4.1 we address the "Anycast Communication and Computation" problem, defined as follows: Given the Grid network we try to find the computation resource to execute a task and the path over which to route the data required by the task. We present two variations to this problem and two, classes of algorithms to address these variations. In Section 4.1.1 we present a multi-constrained routing and scheduling algorithm that uses the path (communication) delay and the cluster (computation) load as selection metrics. The network and the computation resources are co-allocated, in the same time-frame, and thus this algorithm falls in the "task requesting many service guarantees – concurrent reservation" usage pattern. The algorithm presented in Section 4.1.1 is an online algorithm that was designed to function in a centralized scheduling architecture. In Section 4.1.2 we present a multicost algorithm for the joint scheduling of the communication and computation resources needed by a task. 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 or data burst and (ii) the execution of the task at the cluster, defining in this way a simple two-step workflow. 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 scheme can be used for providing network and computation delay guarantees to a task for the simple workflow paradigm defined above. Thus, this algorithm falls in the "Task requesting many service guarantees – workflow" usage pattern. The algorithm presented in Section 4.1.2 is an online algorithm that was designed to operate in a distributed scheduling architecture, but can easily be extended to operate in a centralized manner.

In Section 4.2 we address a problem of concurrent co-allocation of network, storage and computation resources. More specifically, each task is characterized by its processing rate, the size of the input (output) datasets, and the input (output) bandwidth. Also, each task has a budget to be paid when the task is accepted for execution. Scheduling is performed at periodic intervals (offline algorithm, with one phase) in a centralized manner. We present an ILP formulation whose objective is to maximize the profit of the scheduler. Other optimization objectives can also be used. The proposed ILP model incorporates two functions: task admission control and resource-assignment. A solution to the ILP problem tells us whether or not a task is accepted for execution, and if it is accepted, which resources it may use. Accepted tasks are started immediately and at the same time; rejected tasks are transferred automatically to the next scheduling round (i.e., a task will occupy its assigned resources during a number of periods). Thus, this algorithm falls in the "Task requesting many service guarantees – concurrent reservation" usage pattern. In particular, a task requests the concurrent co-allocation of computation, storage and network resources for at least one scheduling period. .

In Section 4.3 we present the MetaScheduling Service (MSS) of the VIOLA project and the extensions required to cope with in-advance reservation of network resources. The MSS developed in the VIOLA project allows the end-user to execute the individual components of his application using the most appropriate resources available. The orchestration of resources located at different sites and belonging to different administrative domains is performed by the MSS. Also, the MMS is responsible for the negotiation, for the resource usage, with the individual local resource management systems, using WS-Agreement. Thus, the presented MSS falls in the "Task requesting many service guarantees – workflow" usage pattern. The MetaScheduling Service developed in the VIOLA project will be enhanced to cope with the requirements of the applications in the Phosphorus project. Finally, we turn our attention to the advance reservation of network resources and present a brief overview of the network reservation system ARGON developed in the VIOLA project. ARGON includes an advance reservation capable interface for the Grid application layer and thus offers connectivity services with a specified QoS on top of the optical network between the Grid sites in the VIOLA network.

1.5 Relation to the Phosphorus Project

Phosphorus' emphasis is on network-related aspects of Grid computing, and thus its focus is on applications that are heavily dependent on communication resources. Although in this deliverable we mainly address scheduling problems on computation resources, we also look at the joint scheduling of communication and computation resources. Several of the algorithms described here are applicable to applications designed to run on the Phosphorus tested. In particular, in deliverable D.3.1 [77] a set of applications designed to run in the Phosphorus project is described. There are five applications, namely: WISDOM, KoDaVis, TOPS, DDSS and INCA, from which three (KoDaVis, DDSS and INCA) have none or little requirements for computation resources, while the remaining two (WISDOM and TOPS) have a significant computation part.

Communication related applications

The KoDaVis application deals with large scale visualizations. Large data sets with a typical size of about 1 terabyte are pre-computed and stored at the super-computer site (not locally at the scientists' lab). A scientist-client wishing to perform a visual analysis, accesses only parts of the data. The collaboration among scientists is enabled through the use of a tele-conference established among them. There is a parallel data-server that distributes fragments of data selected by the clients, and a collaboration server that synchronizes all clients. As KoDaVis is an interactive, collaborative application, it imposes end-to-end delay and delay jitter requirements on the network connectivity.

The DDSS (Distributed Data Storage Systems) applications are widely used to transport, exchange, share, store, backup/archive and restore data in many scientific and commercial applications. The proposed test scenarios include two DDSS use cases: (i) data transfers performed using open-source GridFTP application and (ii) backup/archive/restore operations performed by a commercial application. The communication model is one-to-one or many-to-one. The DDSS applications require small end-to-end communication delay.

The INCA application defines two use-cases: (i) Video on-Demand applications: In these scenarios data can be pre-cached and dispatched on demand. (ii): Bio-Informatic and pharmaceutical research: Collaborative experiments requiring large data sets. Big scientific centers have the storage capabilities to hold the whole data set, while smaller institutions or single researchers cannot afford such storage farms. But even for research centers with sufficient storage capacity, the pattern matching tasks require significant temporary data buffering, which have to be placed at different locations. In both use-cases the key problem is that of planning the placement of data sets on storage resources and then allocating network resources (between storage sites and clients) on demand.

As mentioned above, these three applications pose heavy requirements mainly on communication resources and thus the algorithms proposed in this deliverable are not directly applicable, except for some of the algorithms proposed in Section 4 for scheduling and routing communication tasks. For task routing algorithms please refer to deliverable D5.3 [79].

Applications with significant computation requirements

The WISDOM (Wide In Silico Docking On Malaria) application is a docking workflow/service that allows the researcher to compute millions of compounds of large scale molecular dockings on targets implicated in diseases like malaria (in silico experimentation). There is a pre-staging phase where the software and the input-data are transferred to the sites where the docking simulation will be running (estimated size of data: 0.5 GB) and a post-staging phase, where the output-data are gathered and stored in a common data-base (estimated size: 0.5 TB). Currently, the resource selection is done manually by the user. The WISDOM application can be viewed as a problem of joint allocation of storage, computation and communication resources. The algorithm described in Section 4.2 and the Meta-Scheduling Service described in Section 4.3 could be used to select the resources in an efficient manner. For delay bounds in computation resources the framework described in Section 3 can be employed.

Finally, TOPS is a visualization service for large scale simulation data, similar to the KoDaVis application. However, the difference between KoDaVis and TOPS is that in the former the visualization is performed at the clients' sites while in the latter a remote computation resource, closely located to the data, is used to generate the visualization which is then forwarded to the client. Reservations are made for the graphics (computation) resources, and the network connection between the visualization service and the client (display). Currently, the resource selection is done manually by the user. The problem is that of a concurrent co-allocation of computation and communication resources which is similar to the anycast problem. The algorithm of Section 4.1.1 can be directly used. The algorithm described in Section 4.1.2 has to change in order to function in the opposite order (computation and then communication resources). Moreover, the algorithm described in Section 4.2 and the Meta-Scheduling Service described in Section 4.3 are also applicable. Finally, for delay bounds in computation resources we can use the framework described in Section 3.

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], [14] and [15] 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 missing rate [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 Grid resource scheduling and propose GridIS, a Peer-to-Peer (P2P) decentralized scheduling framework. In GridIS a user (consumer) sends a task announcement via a portal, when he wants to execute that task. The task 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 Network makes the enforcement of fairness in Grids a more complex issue than, for example, in Data Networks. In Data Networks the Generalized Processor Sharing (GPS) [16] 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. Since GPS is difficult to implement, its approximation Weighted Fair Queuing (WFQ) [17] is instead often used in Data Networks (Appendix D). The WFQ exploits concepts of the Max-Min Fair sharing scheme [18]. GPS-based algorithms are widely implemented in the Internet and mobile communications today. In Grid Networks a number of fair scheduling algorithms have also been proposed [19][20].

In this section we present a number of fair scheduling algorithms for Grid Networks. The fair scheduling algorithms that will 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 (the "resource-assignment" phase). These algorithms incorporate fairness considerations in one of these two phases, attempting to share in a fair way the computational capacity of the Grid. Of course the fair sharing of the capacity is influenced by the specific characteristics of the tasks and the users (e.g. task length, task deadline, user budget). The proposed algorithms can run either in a centralized or a distributed scheduling architecture.

In Section 2.1, we propose three new fair scheduling algorithms for Grid computing [24] 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 and then assigns them to the appropriate processors[1] using a modified Earliest Completion Time (ECT)-based policy. The second, called Adjusted Fair Task Order (AFTO) algorithm, refines the SFTO policy by ordering the tasks using the adjusted fair completion times, resulting in more fair treatment of the tasks. Finally, the third scheme we present, called the Max-min Fair Share (MMFS) scheduling 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. All schemes try to assign tasks to processors so as to satisfy their deadline requirements, and if this not possible due to congestion, they assign tasks so that the amount of time by which they miss their deadline is determined in a "fair" way.

In Section 2.2 we propose another scheduling algorithm for Grid, called Fair Completion Time Estimation Algorithm (FCTE) , which takes fairness as a criterion in its scheduling decisions,. This algorithm incorporates with fairness in the second of the two phase scheduling procedure. Under FCTE, all tasks have the same rights in the use of Grid resources. In FCTE tasks are scheduled based on the fair completion time of a task 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. The algorithms we propose provide a tradeoff between fairness and algorithmic complexity.

In Section 2.3 we describe a scheduling procedure that provides fairness among users [82] 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 [25] or in Grid Networks [21], 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

[1] The words "processor " and "computational resource" are used interchangeably in this context

them (a "user" may also refer to a Virtual Organization, or VO, using the Grid infrastructure). For example, it is not fair for a task belonging to a BE user who creates only this task, to be handled equally with the possibly thousands of tasks created by some other BE user. Different weights can also be given to users (or VOs), in which case we talk about weighted user fairness. This for example could be the case in a university campus, where all the BE users-students using the Grid infrastructure deserve the same kind of service from it, while BE users-professors may be given a higher weight. Our proposed schemes for Grid QoS and fairness draw on corresponding schemes and concepts developed for Data Networks.

2.1 Max-Min Fair Scheduling Algorithms

In this and the next section we present three fair scheduling algorithms for Grid computing that take the task deadlines into account [24]. These algorithms incorporate fairness considerations in the first of the two phase scheduling procedure, that is, they determine a fair order in which tasks should be considered for scheduling, and are based on the Max-Min fair sharing concept.

2.1.1 Notation and Problem Formulation

We define the workload w_i of task T_i, *i*=1,2,...,*N,* as the duration of the task when executed on a processor of unit computation capacity, where *N* is the total number of tasks that have 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 some other method. An algorithm for workload prediction of 3D rendering in a Grid architecture is presented in one of our earlier works in [22]. 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 served on a processor 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. (The computation capacity of a processor is the available capacity of the processor, and it does not include capacity occupied by local or system tasks). The total computation capacity *C* of the Grid is defined as:

$$C = \sum_{j=1}^{M} c_j \ . \qquad (1)$$

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 that defines 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 that case, the user may choose not to execute the task, as may be the case when he/she expects the results to be outdated or not useful by the time they are provided. 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 γ_j be the estimated completion time of the tasks already running on or already scheduled on processor *j*. Therefore, γ_j is equal to zero (corresponding to the present time) when no task has been allocated to processor *j* at the time a task assignment is about to be made; otherwise, γ_j is equal to the remaining time until completion of the tasks already assigned to processor *j*. We define the *earliest starting time* δ_{ij} of task T_i on processor *j* as

$$\delta_{ij} = \max\{d_{ij}, \gamma_j\}. \quad (2)$$

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,

$$\delta_i = \frac{\sum_{j=1}^{M} \delta_{ij} c_j}{\sum_{j=1}^{M} c_j}. \quad (3)$$

We will refer to δ_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, δ_i plays a role reminiscent of that of the (mean) memory access time in uni-processor computers.

In the fair scheduling algorithm that we will propose in Section 2.2.4, the *demanded computation rate* X_i of a task T_i will play an important role, and is defined as

$$X_i = \frac{w_i}{D_i - \delta_i}. \quad (4)$$

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 δ_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 fair scheduling algorithms of Sections 2.1.3 and 2.1.7 attempt to reduce the computational rates allocated to different tasks in a fair way.

The scheduling algorithms that we will propose (except for the MMFS algorithm proposed in Section 2.1.7) consist of two phases. In the first phase, we determine the order in which tasks are considered for assignment to processors (the "queuing order") and in the second phase we determine the processor on which each task is scheduled (the "processor assignment").

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 (i.e., the task with the earliest deadline) is served first, followed by the remaining tasks according to their urgency.

The EDF rule answers only the "task-ordering" question, but it does not determine the processor where the selected task is assigned. To answer the "resource-assignment" question, the Earliest Completion Time (ECT) technique presented shortly can be used. The EDF/ECT algorithm is also identical to the Horizon scheduling used in burst switched networks [54].

If task T_i starts execution on processor *j* at the earliest starting time δ_{ij}, its completion time will be $\delta_{ij} + w_{ij}$, where $w_{ij} = w_i / c_j$ is the execution time of task T_i on processor *j*. (Recall our assumption that each task occupies 100% of a processor's capacity when executed; in this way, tasks are on the average executed in the earliest possible time, since time sharing can only increase the average task delay). Among the *M* available processors, the ECT rule selects the processor $\hat{j}$ that minimizes the quantity

$$\hat{j} = \underset{j \in \{1,\Lambda,M\}}{\arg\min} \{\delta_{ij} + w_{ij}\}. \qquad (5)$$

The earliest starting time δ_{ij} depends through Eq. (2) on the time γ_j at which the last task already allocated to processor *j* is expected to complete service.

A note regarding the way γ_j is defined is necessary here. One way is to define γ_j as the processor release time, that is, the time at which all tasks already scheduled on this processor finish their execution. Figure 3 illustrates a scheduling scenario in which a) the task queuing order is selected using the EDF algorithm, b) the processor/resource assignment is selected using the ECT approach and c) γ_j is defined as the processor release time. In this example, we assume that all tasks are available for scheduling at time *t*=0.

[1] In this document the terms "task" and "job" are used interchangeably.

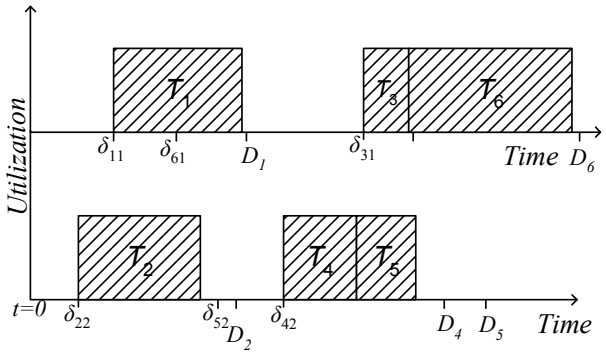


Figure 3 - An example of the Earliest Deadline First/Earliest Completion Time (EDF/ECT) algorithm for the case γ_j is defined as the processor release time. We assume that all tasks T_i, i=1,2,..,6, request service at time t=0. and we have two processors of equal computational capacity (c_1=c_2), which are initially idle. We also assume that D_1<D_2<•••<D_6 and $\delta_{i1} = \delta_{i2}$. The task T_1 with the earliest deadline D_1 is assigned first on processor 1 (chosen randomly in this case since there is tie). Task T_2 is assigned next on processor 2 (since it is the processor that yields the earliest completion time). In a similar way, the remaining tasks are assigned.

Defining γ_j as the processor release time makes it easy to compute, and 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 (for example, the gap between tasks T_1 and T_3 in Figure 3), 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. Among all candidate time intervals the one that provides the earliest completion time is selected. Figure 4 shows how the schedule for the example given in Figure 3 is improved by exploiting capacity utilization gaps. The completion times of tasks T_5 and T_6 are smaller than those of Figure 3. The gap filling version of the algorithm is very similar to the Latest Available Unused Channel with Void Filling (LAUC-VF) adopted in burst switching [55].

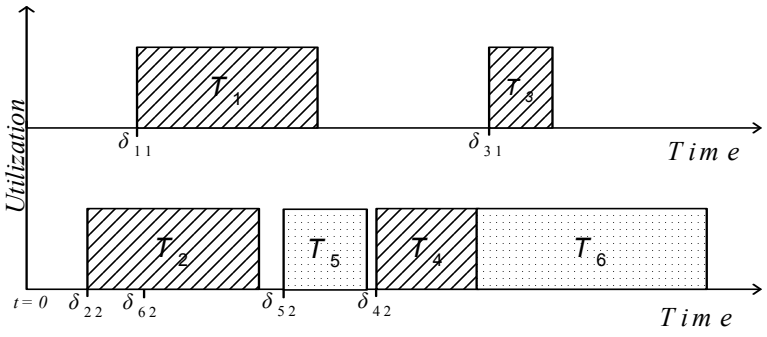


Figure 4 - An example of the EDF/ECT algorithm that exploits processor utilization gaps.

2.1.3 Fair Scheduling

The simple scheduling algorithms described in the preceding section do not adequately address congestion and they 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. With the ECT rule, tasks that have long execution times have a higher probability of missing their deadline even if they have a late deadline. Also, with the EDF rule, a task with a late deadline is given low priority until its deadline approaches, giving no incentive to the user to specify an honest deadline (especially in the absence of any pricing mechanism). To overcome these difficulties, we propose in this section 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 as described in the following subsection. It should be mentioned that the algorithms proposed in this section are oriented towards large scale computing systems consisting of multiple processors.

2.1.4 Estimation of the Task Fair Rates

2.1.4.1 *Ideal Non-weighted Max-min Fair Sharing of Grid Resources*

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 in the Max-Min fair sharing scheme, 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 following example, where four tasks with demanded rates 10, 8, 5 and 15 units, respectively, request service. The iterations involved are shown in Table 1. Let us assume that the 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. Since all tasks are assumed to be of equal importance, the algorithm initially divides the 30 units of the total processor capacity into four equal parts each of 30/4=7.5 units. The second and the third task request service less than 7.5 units (3 and 5 respectively) and thus they get the rate they request. In contrast, the first and fourth task demand rate more than 7.5 units (10 and 15 respectively) and therefore at the first iteration of the algorithm a rate of 7.5 units is assigned to them. Consequently, at the end of the first iteration a residue of 30-23(7.5+3+5+7.5)=7 units is left to be allocated in the following steps. In the second iteration, the residue of 7 units is equally shared among the first and fourth task, so that each of them gets an additional rate of 7/2=3.5 units. Since, however, the first task request service less than 11 (=7.5+3.5) units, it gets the rate it requests, i.e., 10 units. The fourth task gets a rate of 11 units at the end of the second iteration and a residue of 1 unit is obtained. This residue is then given to the fourth task whose rate is less than the demanded one in the third iteration of the algorithm. At the end of the non-weighted max-min fair sharing algorithms, the non-adjusted fair computational rates r_i of tasks T_i are computed. A graphical illustration of the aforementioned example is depicted in Figure 5.

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. The overall processor capacity is taken to be equal to 30 units.

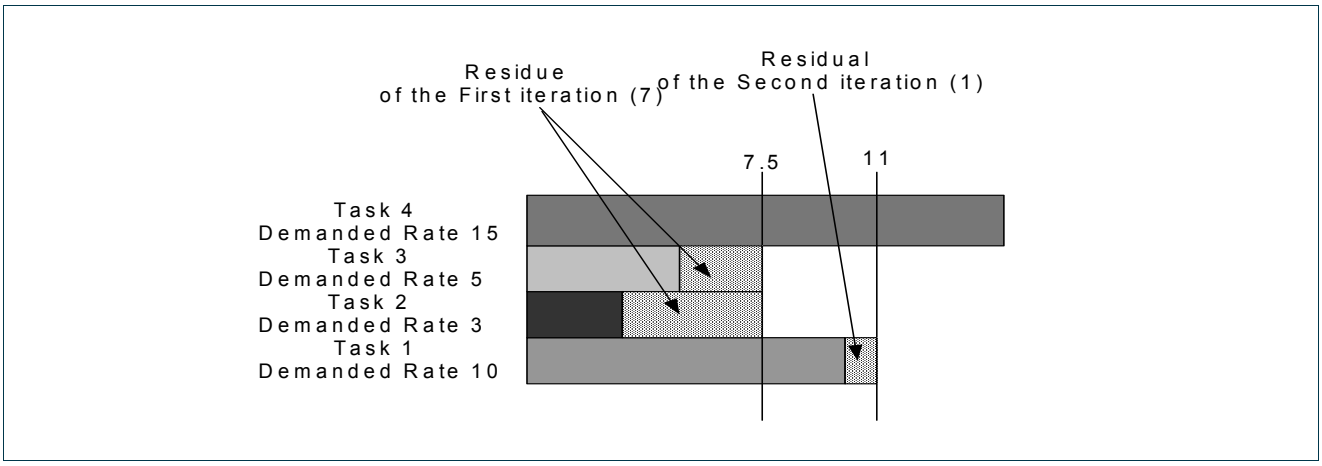


Figure 5 - A graphical conceptualization of the examples of Table 1.

More details about the max-min fair sharing algorithm can be found in Appendix A.

2.1.4.2 *Ideal Weighted Max-min Fair Sharing of the Grid Resources*

We now consider the case where users have different priorities. More specifically, we assume that each task T_i is characterized by an integer weight φ_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. We assume, without loss of generality, that the smallest task weight is equal to one.

In weighted Max-Min fair sharing scheme the rate a user with weight φ_i gets is equal to the total rate of φ_i simple users in the (non weighted) Max-Min fair sharing scheme.

A detailed description of the weighted Max-Min fair sharing algorithm is presented in Appendix B.

2.1.5 Fair Task Queue Order Estimation

As mentioned previously, a scheduling algorithm should make two decisions. First, it has to choose the order in which the tasks are considered for assignment to a processor ("task-ordering"). Second, for the task that is located each time at the front of the queue, the scheduler has to 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 2.1.5.2 and 2.1.5.3, several ordering disciplines of different degrees of implementation complexity. Before doing so, however, we introduce some notation that will be useful in our presentation.

2.1.5.1 *Non-adjusted Fair Completion Time Estimation*

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

$$t_i = \delta_i + \frac{w_i}{r_i}, \qquad (6)$$

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 δ_i (recall that δ_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 δ_i. More precisely, a) the task is actually assigned to a specific processor j and the earliest starting time on that processor is δ_{ij}, b) even if $r_i < c_j$, it may not be possible to execute the task at rate r_i on that processor, since we assume that time sharing is not supported, c) the estimates w_i of the task workloads may be inaccurate. The non-adjusted fair completion times t_i do not correspond to realistic completion times, but are used by our algorithm merely as an index for determining the order in which tasks are processed by the scheduler.

2.1.5.2 *Simple Fair Task Order (SFTO) scheme*

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. (6). 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. The SFTO rule is simple to implement, but it is not as close an approximation to the max-min fair concept as some of the other rules to be described in the following. Its performance and fairness characteristics are, however, rather good as the simulation results presented in Section 2.1.10 wll indicate.

It should be mentioned that in the proposed fair scheduling algorithms, the task fair rates are approximately estimated by taken into consideration the total offered capacity of all M processors. However, the task assignment exploits the properties of each individual processor resulting in a multiprocessor schema.

2.1.5.3 *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 and in the SFTO scheme presented in Section 2.1.5.2 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. Also, 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 sections, 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 approximate 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. The way the adjusted fair completion times can be computed is described next. Simulation results on the performance and computation complexity of all schemes will be presented in Section 2.1.10.

To compute the adjusted fair completion times t_i^a, the fair rate of the active tasks at each time instant must be estimated. This can be done in two ways. In the first approach, each time unused processor capacity is assigned, it is equally divided among all active tasks. In the second approach, the rates of all active tasks are re-calculated using the max-min fair sharing algorithm, as described in Section 2.1.4, based on their respective demanded rates. The first approach is considerably less computationally intensive than the second one, since the max-min fair sharing algorithm is activated only once. The second approach, however, yields a schedule that is fairer. Regardless of the approach used, the estimated fair rate of each task is a function of time, denoted by $r_i(t)$.

Having estimated the fair rates $r_i(t)$, the fair completion time can be obtained using the following algorithm. We assume that the rates $r_i(t)$ of all tasks have been normalized so that the minimum fair task rate equals 1. We introduce a variable called the round number, which defines the number of rounds of service that have been completed at a given time [23]. A non-integer round number represents a partial round of service. The round number depends on the number and the rates of the active tasks at a given time. In particular, the round number increases with a rate equal to the sum of the rates of all active tasks, i.e., with a slope equal to $1 \Big/ \sum_i r_i(t)$. Thus, the rate with which the round number increases changes and has to be recalculated each time a new task arrival or task completion takes place.

Based on the round number, we define the finish number $F_i(t)$ of task T_i at time t as

$$F_i(t) = R(\tau) + \frac{w_i}{r_i(t)}, \qquad (7)$$

where τ is the last time a change in the number of active tasks occurred (and, therefore, the last time the round number was recalculated), and $R(\tau)$ is the round number at time τ. $F_i(t)$ is recalculated each time

new arrivals or task completions take place. Note that $F_i(t)$ is *not* the time that task T_i will complete its execution but is only a service tag used for ordering the tasks. Using Eq. (7), the adjusted fair completion time t_i^a can be computed as the time at which the round number reaches the estimated finish number of the respective task. Thus,

$$t_i^a : \quad R(t_i^a) = F_i(t_i^a) . \tag{8}$$

As mentioned earlier, the task adjusted fair completion times determine the order in which the tasks are considered for assignment to processors in the AFTO scheme: the task with the earliest adjusted fair completion time is assigned first, followed by the second earliest, and so on.

2.1.6 Fair 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). According to this rule, each task is assigned to the processor that yields the earliest completion time. Simulation results on the performance of the SFTO and AFTO schemes when combined with the ECT rule are described in Section 2.1.10.

2.1.7 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

$$\min E = \min \sum_{i=1}^{N} \left| r_i^s - r_i \right| \tag{9a}$$

subject to

$$\sum_{i \in P_j} r_i^s \le c_j \qquad P_j = \{i : T_i \text{ scheduled on } j \text{ processor}\} \tag{9b}$$

The set P_j contains all tasks scheduled on processor *j*. The total deviation *E* of the schedulable rates from the fair rates will be referred to as the error of the scheduler.

The minimization of Eq. (9a) subject to the constraint of Eq. (9b) can be found using the algorithm described in Appendix C. The main idea of the proposed scheme is to perform an initial processor assignment and then, appropriately re-arrange underflowed with overflowed processors so that a better exploitation of the processor capacity is obtained. This is illustrated in the example of Figure 6 where we consider two processors of capacities 20 and 25 units, and six tasks with fair rates 2, 5, 5, 6, 9, 10 units that have to be scheduled. Initially, processor 1 is overflow (its capacity is 20 units and the sum of the rates of the assigned tasks is 21 units), while processor 2 is underflow (its capacity is 25 units and the sum of the rates of the assigned tasks is 16 units). By rearranging the task of rate 2 initially assigned to processor 2 with the task of rate 10 initially assigned to processor 1, both processors turn to the underflow state, resulting in a reduction in the error. The example of Figure 6 illustrates one iteration of the algorithm; in full algorithm implementation more iterations take place.

The MMFS scheme assigns tasks to processors so that their actual scheduled rates are as close as possible to their respective fair rates, but it does not guarantee that a feasible solution is found, i.e., it does not guarantee that all tasks are assigned to the available processors without any violation of the respective deadlines. As a result, a rate reduction is required for those tasks that are assigned to overflow processors, so as to achieve a feasible solution. In Section 2.1.9, a fair rate reduction is described to obtain a feasible solution is obtained.

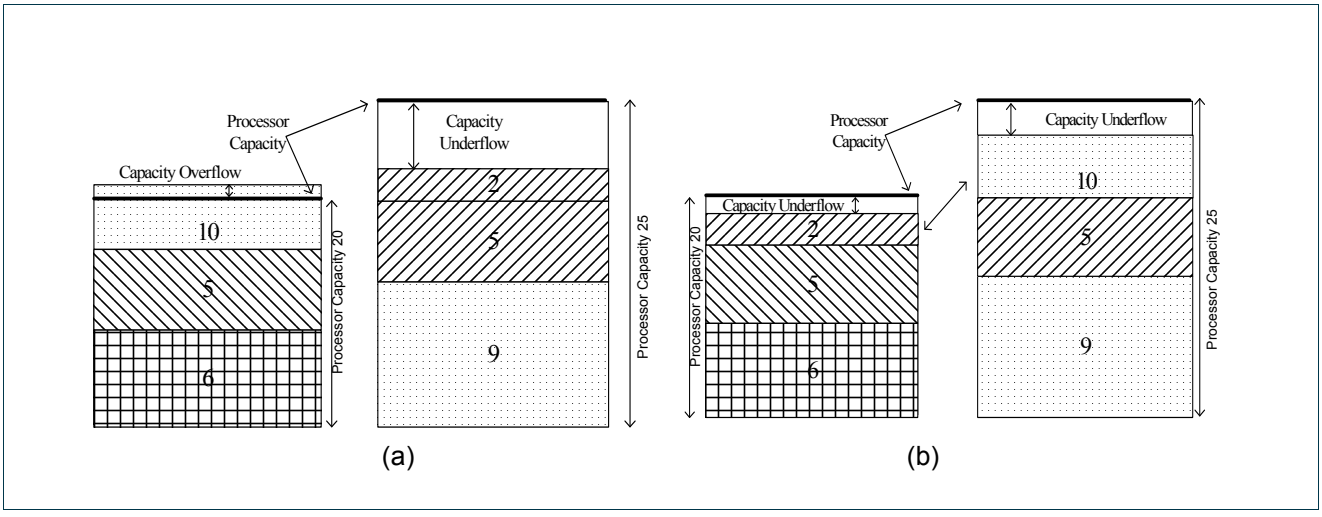


Figure 6 - The idea of the task re-arrangement. (a) Initial task assignment. (b) Task re-arrangement.

2.1.8 Initial Tasks Processor Assignment

For the MMFS scheme to work well, we have to start with a good initial assignment. In what follows, we present a method similar to a heuristic algorithm used in the bin packing problem, which is a well-studied problem in the literature.

Initially, the algorithm sorts the tasks with respect to their fair rates in a descending order. To obtain an initial assignment, the task with the largest fair rate is first assigned to a processor, followed by the task of the second largest fair rate, and so on. The algorithm assigns, if possible, the task to a processor of adequate available capacity. If more than one processors of adequate available capacity exist, we choose the one that would leave the smallest residual capacity after the task assignment is made. In case a selected task cannot be feasibly scheduled on any processor, the task is assigned to the processor of minimal overflow. This process is terminated when all tasks have been scheduled on the available processors.

2.1.9 Fair Sharing of Overflow Capacity and Task Queuing Order

In the previous section, we have described an algorithm for the assignment of tasks to processors so that the task schedulable rates are as close as possible to their respective fair rates. The solution obtained however is not necessarily feasible, since some processors may be overflow. For this reason, the schedulable task rates r_i^s of the overflow processors are reduced in a fair way in order to obtain a feasible solution. In this way, tasks assigned to overflow processors will not be able to meet their deadline, but the amount by which they will miss their deadline will be determined in a fair way.

After finding the schedulable task rates (the solution also gives the processor each task is assigned to), the tasks are scheduled for execution in an ascending order of their fair completion times on the processor to which they have been assigned. That is, the task with the earliest fair completion time is first scheduled for execution, followed by the second earliest task, and so on.

2.1.10 Performance Results

In this section we evaluate the performance of the Grid scheduling algorithms presented in previous subsections. In particular, Section 2.1.10.1, describes several criteria of interest for measuring the performance of the proposed scheduling algorithms, while Section 2.1.10.2 presents simulation results and comparisons of the proposed algorithms with traditional scheduling policies.

2.1.10.1 *Arrival Model*

We first describe the arrival model used in our simulations. 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:

$$\rho = \frac{\sum_{i=1}^{N} X_i}{C}. \qquad (10)$$

From Eq. (10), it is clear that a Grid with load ρ is able to serve, on average, *N* tasks of workload w_i, deadlines D_i and ready times δ_i within a time interval of $\Xi = \rho * E\{D_i - \delta_i\}$, where the $E\{\cdot\}$ denotes the expected value. In the arrival model adopted, we assume that the *N* tasks arrive in the Grid into *β* groups, each of N/β tasks. We also assume that the probability of each group of N/β tasks to arrive in the Grid follows the Poisson distribution with parameter *λ*, where $\lambda^{-1} = \beta \cdot \Xi$ (Batch Poisson arrival model). In our experiments *β*=10. Figure 7 provides and illustration of the adopted arrival model.

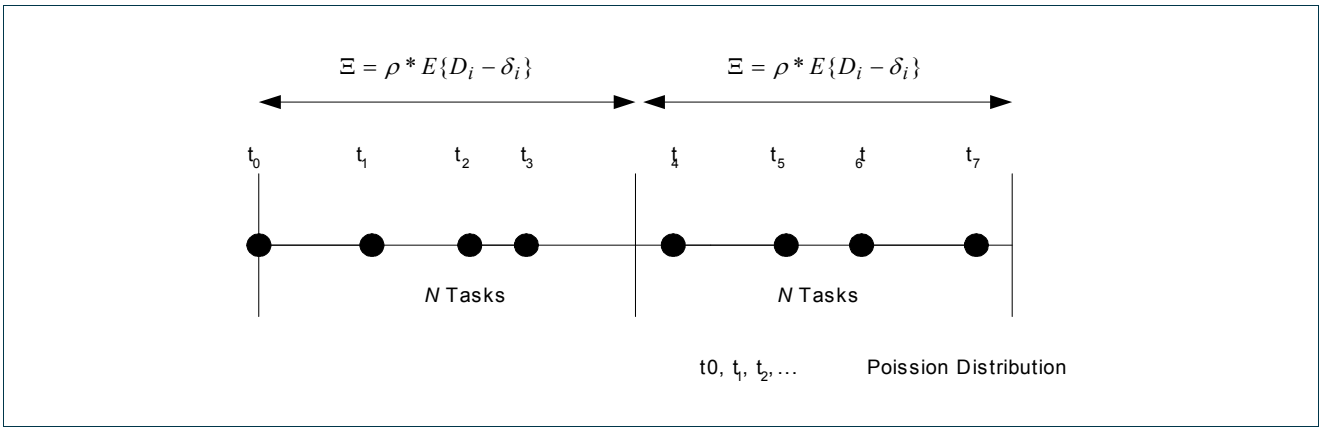


Figure 7 - An illustrative example of the adopted arrival model.

2.1.10.2 *Objective Evaluation*

A criterion we will use for measuring the performance of a scheduling algorithm is the relative error between the demanded task rates and the actual schedulable rates, defined as

$$E_1 = \sum_i \frac{\left| X_i - X_i^c \right|}{X_i}, \qquad (11)$$

where X_i is the demanded rate and X_i^c is the actual rate allocated to the i^{th} task. Low values of error E_1 indicate that most of the tasks are served at rates close to their demanded rates.

In the FCFS and EDF algorithms, the tasks are either executed at their demanded rates X_i, or they are rejected. Therefore, for the FCFS and EDF schemes, the actual task rates $X_i^c \in \{X_i, 0\}$, depending on whether or not the task is accepted for execution. In contrast, in the fair scheduling schemes we proposed, all tasks are executed, possibly at a rate smaller than their demanded rate. Execution of a task with a rate smaller than its demanded rate means that the task deadline is violated.

Another criterion we will use for comparing the performance of the scheduling schemes is the ratio

$$E_2 = \frac{\sum_i X_i^c}{C}. \quad (12)$$

E_2 expresses the efficiency of the scheduling algorithm in allocating the available processor capacity; the greater the value of E_2, the better is the scheduling efficiency. When $\sum_i X_i > C$, an ideal scheduler would use the total offered processor capacity and E_2 would equal 1. When $\sum_i X_i < C$, an ideal scheduler would serve all tasks with rates equal to the demanded ones. In practice, however, due to task and processor constraints (tasks are non-preemptable, time sharing is not allowed, and so on), the ideal case cannot be achieved.

A third criterion we will use for evaluating scheduling efficiency is the average relative deviation of the demanded task deadlines to the actual task completion times,

$$E_3 = \frac{1}{N}\sum_i \frac{\left|D_i - \max(D_i^c, D_i)\right|}{D_i}, \quad (13)$$

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_3.

As already mentioned, 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_3 becomes equal to infinity. To overcome this difficulty, we evaluate the performance of these schemes assuming that tasks whose deadline is violated are put in a waiting list, and reapply for execution after the completion of the last feasibly assigned task.

2.1.10.3 *Simulation Results*

In this section, we simulate the proposed scheduling schemes (SFTO, AFTO and MMFS) against a) a large set of tasks of varying size and workload variance, b) a large and varying number of processors and c) processor asymmetries (e.g., groups of processors of different capacities). Furthermore, in the simulation results, we evaluate the effect the variation of task deadlines has on the performance of the proposed algorithms. In all experiments, the arrival model, described in Section 2.1.10.1, is adopted.

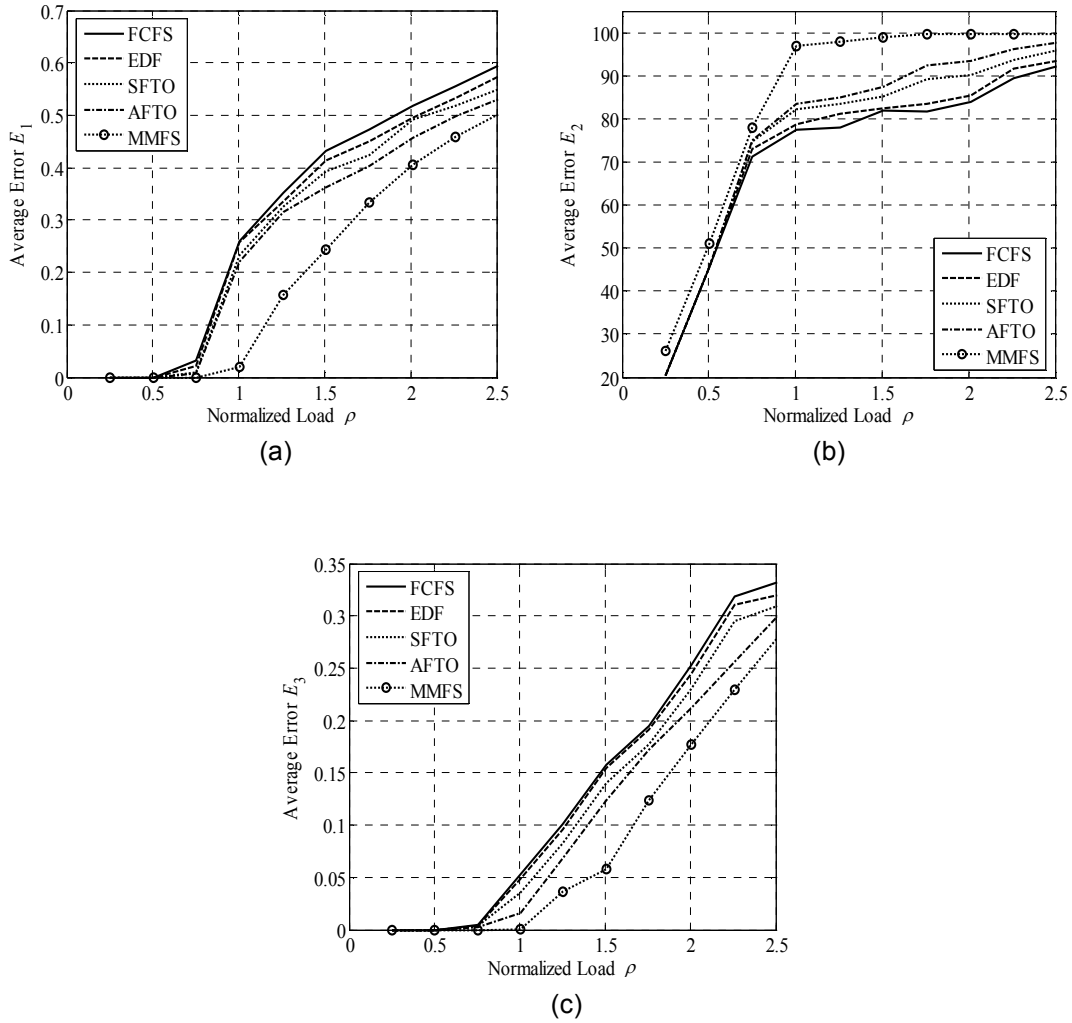


Figure 8 - The errors E_1, E_2 and E_3 versus the normalized load ρ for the FCFS, the EDF, the SFTO, the AFTO and the MMFS algorithms.

The Effect of the Task Size

Figure 8 illustrates the errors E_1, E_2 and E_3 obtained for the SFTO, AFTO, and MMFS scheduling policies against the normalized load ρ (Eq. 5). For comparison purposes, we also depict in Figure 8 the results obtained for the FCFS and EDF schemes. The simulations were performed assuming a Grid consisting of 500 processors of almost the same 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 (see Section 2.1.10.1). The experiment is repeated for 20 time intervals Ξ, i.e., for a total of 2500*20=50000 tasks. In this experiment, we assume that all tasks present almost similar deadlines with a normal distribution of 1% standard deviation of the respective mean deadline value. The mean value of tasks' workload (task size) varies, so that

the normalized load ρ takes values from 0.25 to 2.5, while the respective standard deviation equals 10% the mean value.

Under all criteria, 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 ($\rho < 0.6$), all algorithms efficiently schedule the tasks, but as the load ρ increases (i.e., the task sizes increase), the MMFS policy outperforms the other schemes. As is observed in error E_2, the MMFS performance is close to an ideal scheduler even for heavy load $\rho > 1.5$ (corresponding to congestion).

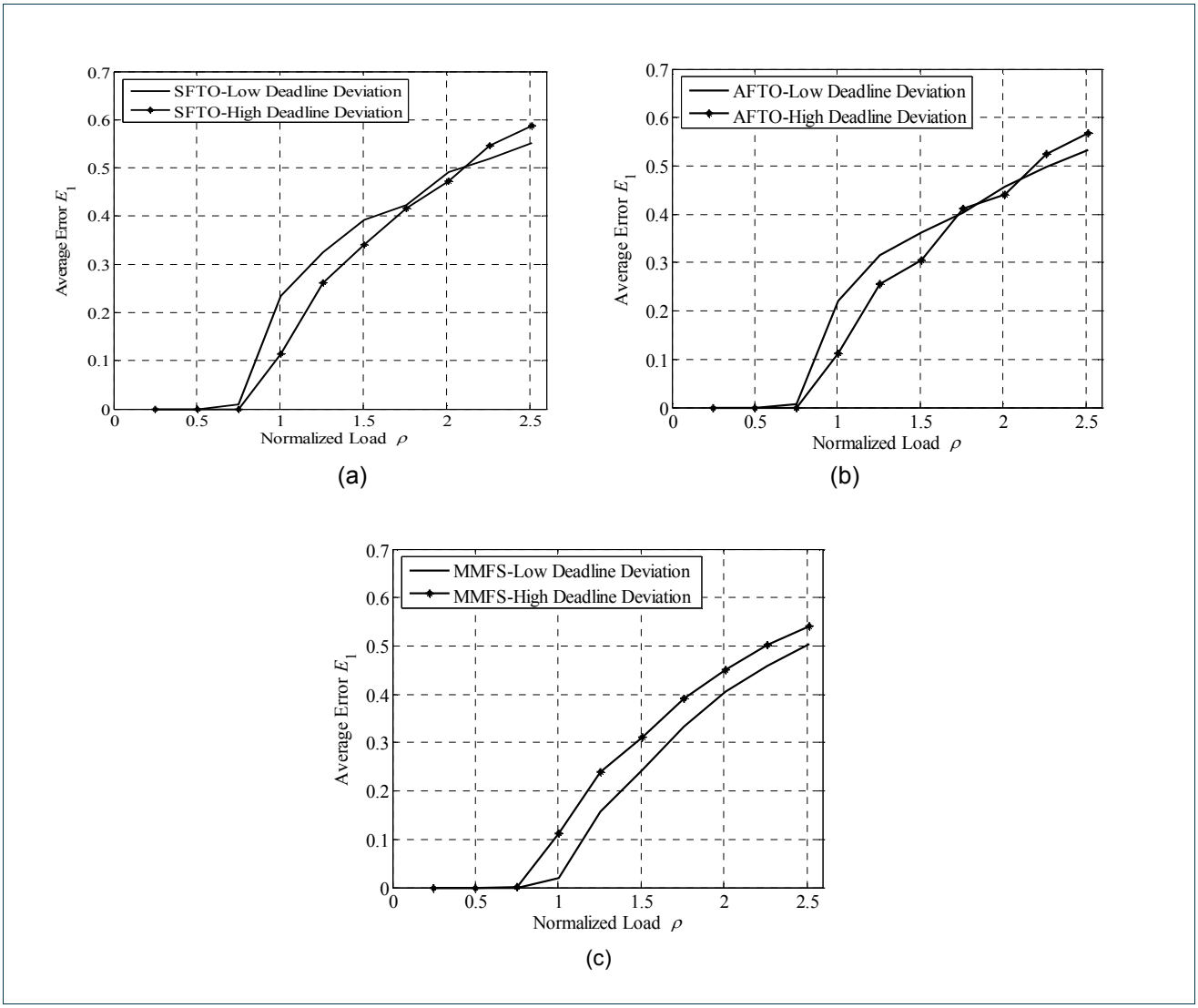


Figure 9 - The error E_1 versus the load ρ in case of low and high deadline variation for a) the SFTO, b) the AFTO and c) the MMFS policy.

The effect of the tasks' deadline variation on the error E_1 for different values of the mean task size is shown in Figure 9. In this figure, we compare the performance of the SFTO, AFTO and MMFS algorithms for low and high values of the tasks' deadline variation. In particular, Figure 9 compares the results obtained from Figure 8(a) with the results obtained assuming that the deadlines follow a normal pdf of high standard deviation (in this experiment, we select the standard deviation to be equal to the mean value of the task deadlines). As is observed, for all algorithms except for MMFS, an improvement in the error E_1 is noticed. However, the MMFS scheduling efficiency deteriorates as the tasks' deadline variation increases meaning that the performance improvements obtained by using the MMFS algorithm compared to the other techniques also decrease. This is because the MMFS policy re-allocates the tasks in the processors without taking into consideration their deadlines. Similar conclusions are drawn using the criterion E_2 (see Figure 10).

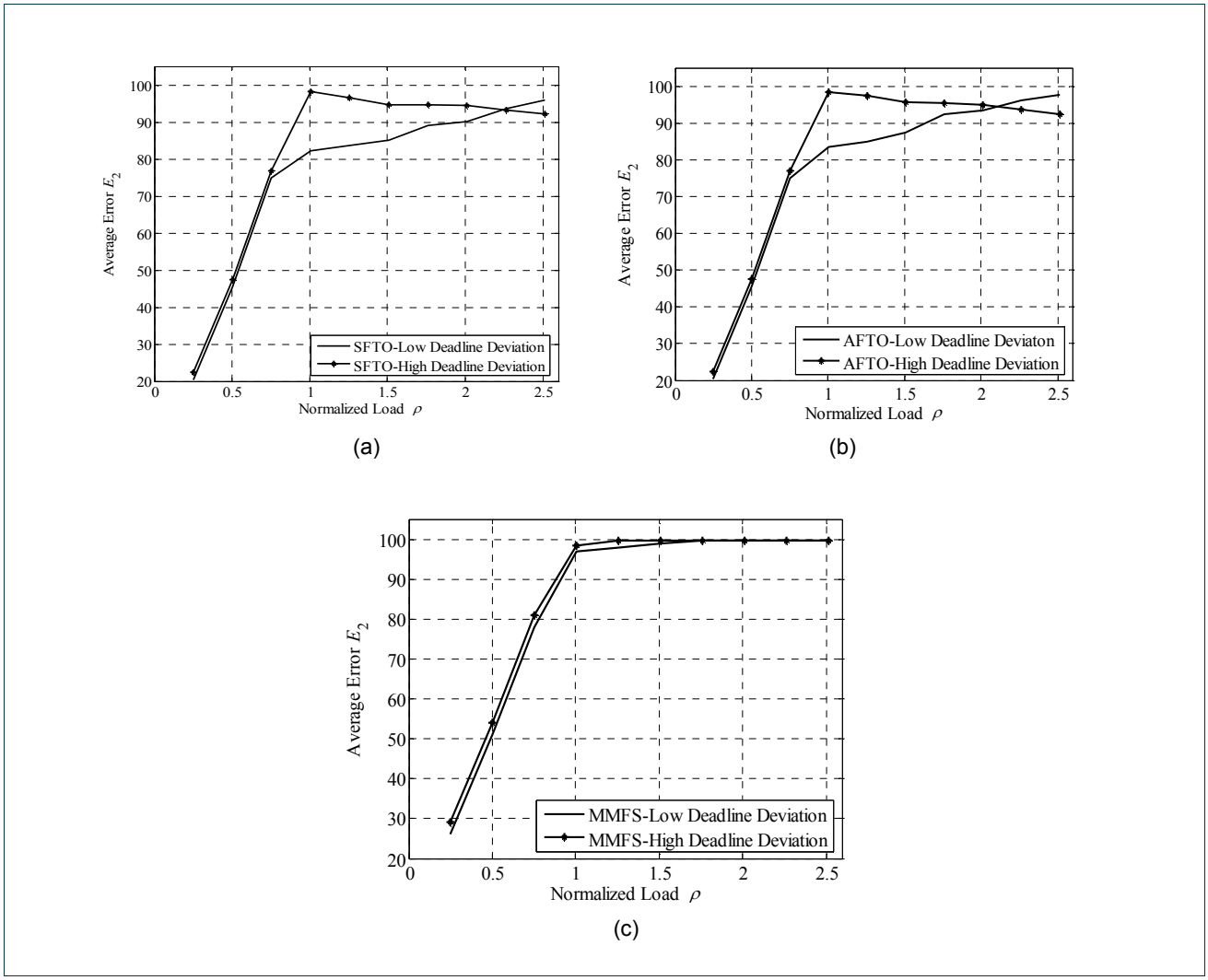


Figure 10 - The error E_2 versus the load ρ in case of low and high deadline deviation for a) the SFTO, b) the AFTO and c) the MMFS policy.

The Effect of Workload Variance

The effect of the task workload variance on scheduling performance is illustrated in Figure 11, where again a symmetric case of 500 processors is considered, tasks have similar deadlines, and $\rho = 1.25$. The arrival model of Section 2.1.10.1 is adopted with 2500 tasks generated within each time interval Ξ. The experiment is repeated for 20 time intervals Ξ, for a total number of 2500*20=50000 tasks. For each experiment, 70 runs have been conducted as in the previous case and the average value over all runs is depicted in Figure 11. The MMFS scheme performs better than the other scheduling methods, for all values of the workload variance and all metrics E_1, E_2, E_3. As the workload variance increases, the performance of all the schemes improves (except for that of MMFS, which remains almost the same since it is close to the ideal case).

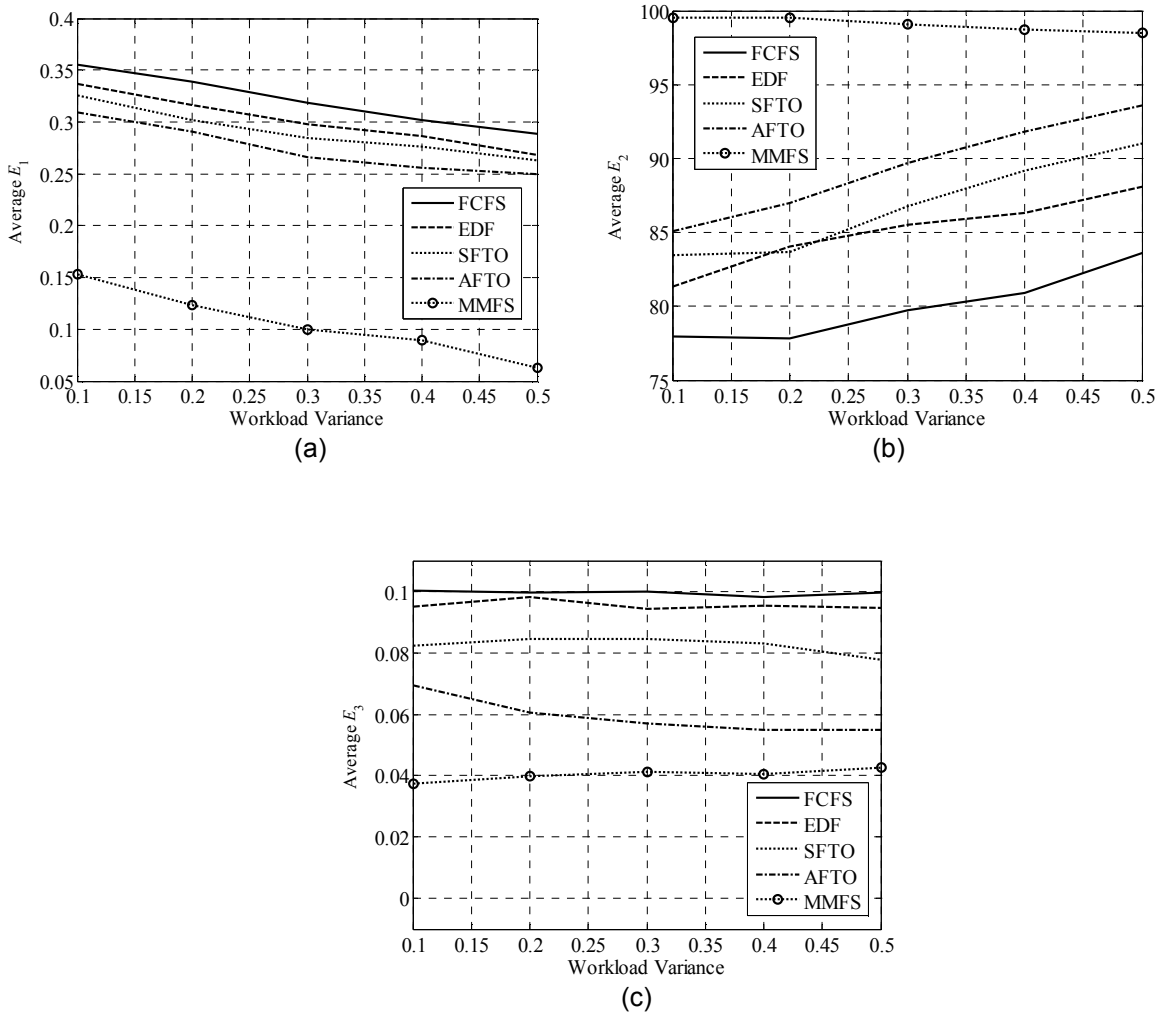


Figure 11 - The errors E_1, E_2 and E_3 versus the variance of the task workload for the FCFS, EDF, SFTO, AFTO and the MMFS scheduling policies.

The Effect of the Number of Processors

Figure 12 illustrates the performance of the five examined scheduling policies with respect to the number of processors. The experiments have been conducted for a number of processors ranging in [50,…,1000]. In all cases, symmetric processors are assumed, and the load is ρ=1.5. It is also assumed that the workload variance equals 10% of the respective mean value. As the number of processors increases, the number of tasks also increases to retain a constant load to the Grid infrastructure. We observe that the MMFS scheduling 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, with a decreasing, however, ratio. For the error E_2, the performance of the MMFS algorithm is close to the ideal one, independently of the number of processors comprising the Grid infrastructure.

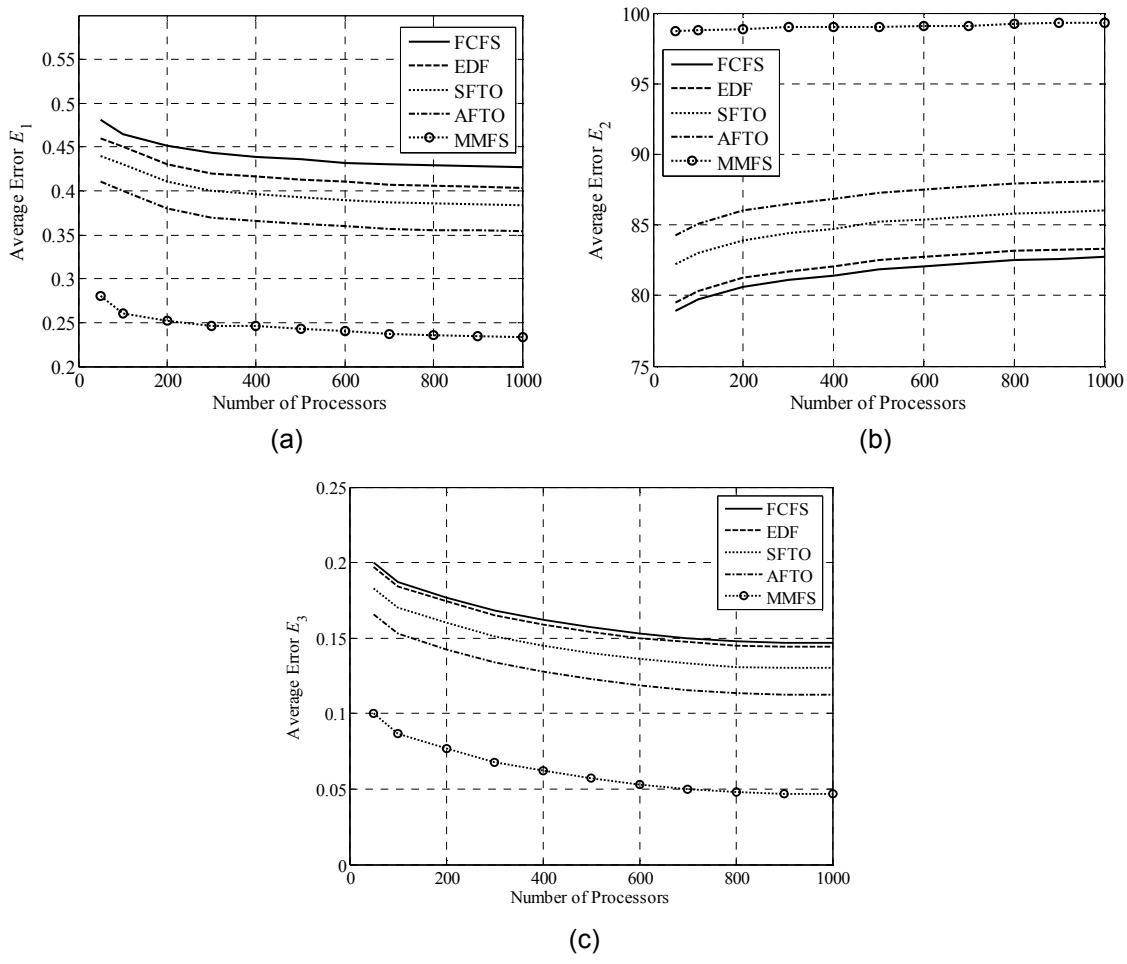
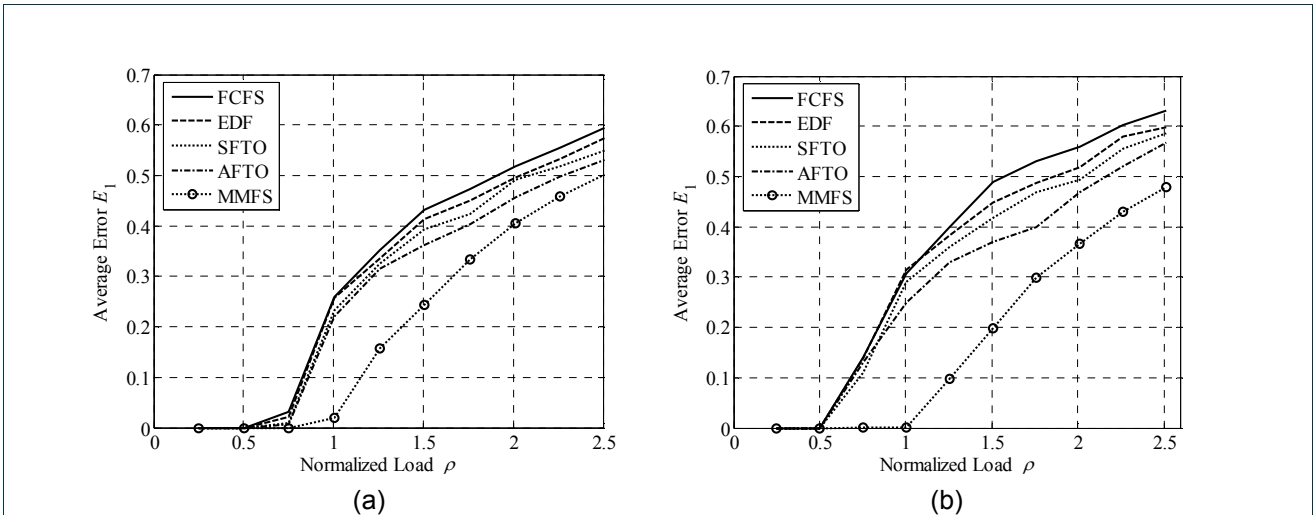


Figure 12 - The errors E_1, E_2 and E_3 versus the number of processors for $\rho = 1.5$ and workload variance 0.1.

The Effect of Processor Capacity Variation

In this section, we evaluate the effect of processor asymmetries (variation of the processor capacity) on the scheduling performance. In all experiments, 70 runs have been conducted and the average value over all runs is depicted. The Grid consists of 500 processors of varying capacity. Furthermore, we assume that 2500 tasks arrive to the Grid within a time interval of Ξ and each experiment is repeated for a total time $20*\Xi$, and a total of 2500*20=50000 tasks.

Figure 13 illustrates the error E_1 for the five examined scheduling schemes as a function of the normalized load ρ under different scenarios for processor capacity. In all experiments, the same mean processor capacity is maintained. Figure 13(a) shows the results obtained when processors have similar capacities (as in Figure 8(a)), while Figure 13(b-f) shows the results for asymmetric processor capacities. In particular, Figure 13(b) assumes that the capacity of the processors follows a normal distribution of high standard deviation (in this case, equal to one half of the respective mean value); this is called Asymmetric Gaussian case. Figure 13(c) assumes a uniform distribution of the processor capacity in the interval of [*a,* 2**μ*+*a*], where *a* is a small value of processor capacity, while *μ* the mean processor capacity value; this is called Asymmetric Uniform case. The case where we have different groups of processors, with processors in each group having the same capabilities, is presented in Figure 13 (d-f). In the scenario of Figure 13(d), the processors are equally divided into two groups, one of high capacity and one of low capacity; this is referred to as the Cluster Symmetric case. Figure 13(e) presents the scenario where the majority of the processors (80%) are of low capacity while the remaining of high capacity; this is called Cluster Biased Low case. Instead, Figure 13(f) shows the opposite scenario, where the majority of the processors (80%) are of high capacity, which is referred to as the Cluster Biased High *case*.



(a) (b)

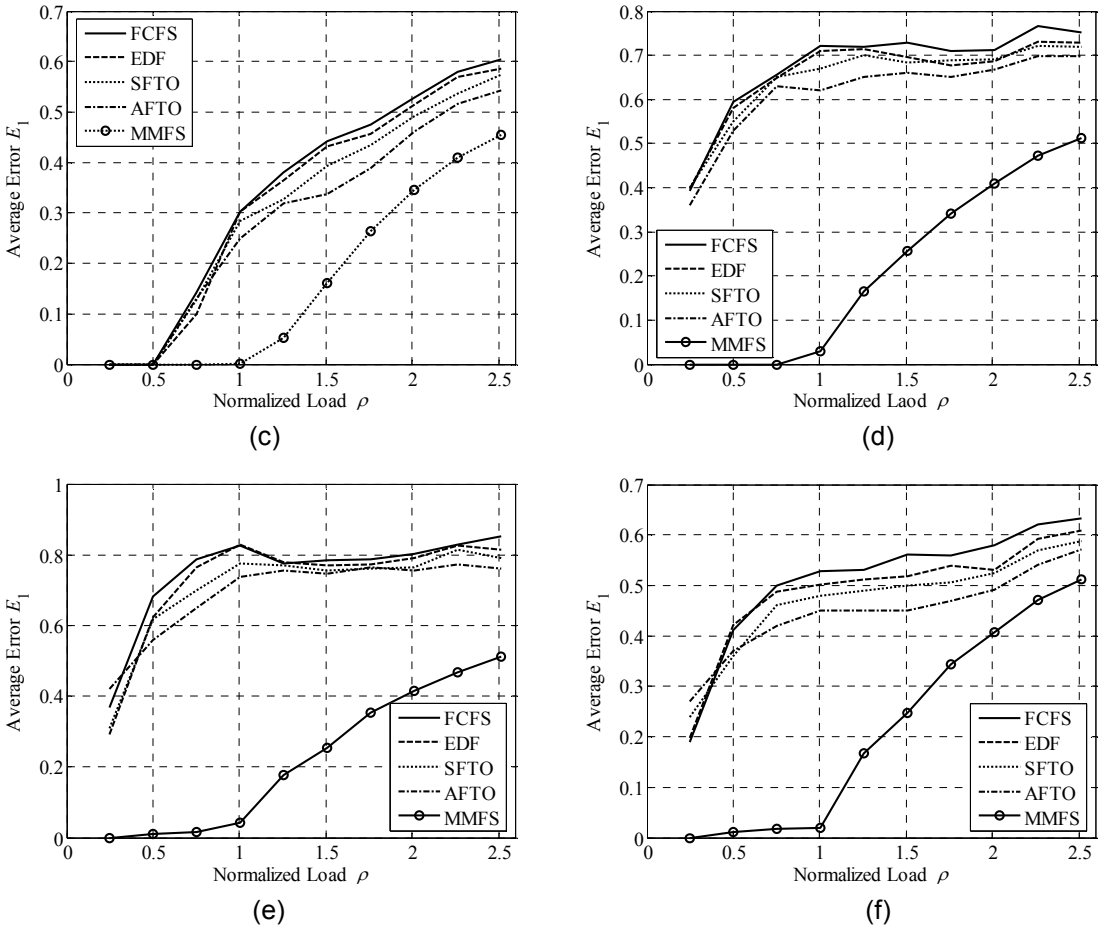


Figure 13 - The effect of processor capacity variation on the error E_1, a) symmetric case, b) asymmetric Gaussian case, c) uniform case, d) cluster symmetric case, e) cluster asymmetric case biased of low processor capacity, f) cluster asymmetric case biased of high capacity.

We observe that in all cases the MMFS algorithm again outperforms all the other examined scheduling schemes. We also note that high deviation of processors capacities deteriorates the scheduling performance of the FCFS, EDF, SFTO and AFTO policies, while the performance of the MMFS scheme remains robust. This is because the task re-allocation used in the MMFS method optimizes the task to processors assignment, something which is not done by the other approaches. This is more evident in the case of many low capacity processors, since the few processors of high capacity cannot compensate the infeasible scheduling of tasks assigned to low capacity processors.

2.2 Fair Completion Time Estimation Algorithm (FCTE)

In this section we propose a new scheduling algorithm for Computational Grids, called Fair Completion Time Estimation (FCTE) algorithm [83] that takes into account fairness considerations in assigning tasks to resources. FCTE is a two phase scheduling algorithm designed for the case where tasks have no deadline or other QoS requirements, and the only objective is to treat tasks in a fair way while maximizing resource utilization efficiency. In the first phase ("task-ordering"), the order in which the tasks will be considered is decided, while in the second phase ("resource-assignment") the ordered tasks are assigned to the resource that minimizes the estimated fair completion time. The fair completion time of a task on a certain resource 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 a time-sharing discipline is used.

2.2.1 Grid Network Model

We consider a Grid network that consists of a number of users, a number of resources and a central meta-scheduler. In our model, we assume that all these entities are directly connected to each other through virtual links (that may actually correspond to paths) of various capacities, so that the details of the routing algorithm employed do not need to be considered. Every task has a non-critical deadline. If a non-critical deadline expires the task remains in the system to complee execution bu it is recorded as a dadine miss. Also every task is characterizedby its length (or workload), defined as the amount of time needed to execute the task on a computing resource of computing capacity equal to 1 unit. Users generate atomic (indivisible) tasks with varying characteristics (workload, deadline) and different inter-arrival times. The central meta-scheduler maintains a queue where it collects task requests until the scheduling decision is made. Each resource site contains a number of machines equipped with a number of CPUs. Also every resource has a local queue, where the arriving tasks are stored, and a local scheduler that schedules the tasks in the local queue to the available machines and CPUs.

Whenever a user creates a new task, he immediately sends information about the task's characteristics to the central meta-scheduler, in the form of a task request. The central meta-scheduler receives task requests by several users and periodically executes the "task-ordering" and the "resource-assignment" phases. Following that, the scheduler returns the assignment decisions back to the users, which send their tasks to the selected resources for execution. The tasks that arrive in a resource are stored in the resource's queue. We assume that a local scheduler assigns tasks from the local queue to free machines and CPUs using First Come First Served (FCFS) scheduling policy. When a task completes, the resource returns its execution results back to the originating user.

2.2.2 Fair Completion Time Estimation Scheduling Algorithm

As mentioned earlier, the Fair Completion Time Estimation (FCTE) scheduling algorithm follows a two-phase procedure. In the first phase the task requests, waiting in meta-scheduler's queue, are ordered using some queuing discipline, while in the second phase tasks are assigned to resources according to their estimated fair

completion times. We assume that the FCFS queuing discipline is used in the ordering phase, however any other ordering discipline may be used (e.g., EDF, LLF, etc). The estimation of a task's fair completion time is performed using one of the following two methods.

2.2.2.1 *Non-adjusted Fair Completion Time Estimation*

The non-adjusted fair completion time is the completion time of a task in a certain resource, assuming that it gets a fair share of that resource's computational power. By fair share we mean that the task is getting $1/N_j+1$ of the *j* resource's computational capacity c_j, as if time-sharing was used. N_j is the total number of tasks already assigned to the resource *j* when the assignment decision is made (Figure 14).

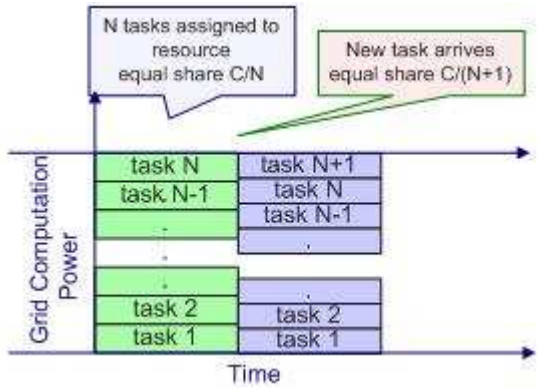


Figure 14 - Non-adjusted fair completion time estimation.

So the fair completion time estimation of a task *i* in a resource *j* is defined as

$$d_{ij} + Q_{ij}\,, \tag{14}$$

where d_{ij} is the time it takes for the data required to execute task *i* to reach resource *j*, and Q_{ij} is the task's *i* fair execution time on resource *j*, defined as

$$Q_{ij} = \frac{\phi_i \cdot (N_j + 1)}{c_j}\,, \tag{15}$$

where φ_i is the weight of task *i*. After the fair completion time of a task *i,* at every resource, has been estimated, the task is assigned to the resource *j* that gives the minimum fair completion time estimation

$$Q_i = \min_j (d_{ij} + Q_{ij})\,. \tag{16}$$

The proposed method for the assignment of tasks to resources is both realistic and easy to implement since it does not require any a-priori knowledge of task workloads.

2.2.2.2 *Adjusted Fair Completion Time Estimation*

It is important to note that the fair processing time Q_{ij} of task *i* on resource *j*, given by Eq. (15), is only an approximation. As new tasks may be sent to resource *j*, or existing tasks may complete execution, the fair share of task *i* in the computational capacity of resource *j* is constantly changing. In this section we define the adjusted fair completion time Q_{ij}^{a}, as the fair completion time of a task *i* in resource *j,* taking into account not only the number N_j of tasks already assigned to the resource *j* but also their completion time estimations. However, we should note that even by using Q_{ij}^{a} we still do not get the exact value of the fair completion time of a task *i* in resource *j*, since there is no way we can account for future tasks that may be sent to the resource.

Figure 15 presents graphically the reason the Q_{ij}^{a} estimation can help us in making better scheduling decision than the Q_{ij} estimation. Figure 15 illustrates two resources *j* and *j'* of equal computation capacity $c_j = c_{j'}$. There are *N* task assigned to each resource, but the tasks assigned to resource *j* have larger workloads. The first task that completes its execution at resource *j* finishes at time t_1 while the first task that completes its execution at resource *j'* finishes at t_1' and $t_1 > t_1'$, for the second task is the corresponding finish times are $t_2 > t_2'$, and so on. In this example we assume that there are no arrivals of new tasks, so the last tasks utilize the whole resource capacity and finish their execution sooner at the resource *j'* than at *j*, since $t > t'$.

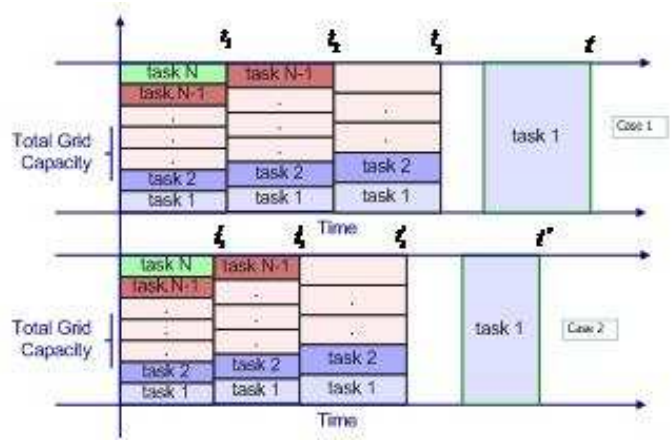


Figure 15 - Adjusted fair completion time estimation.

The example of Figure 15 indicates that the adjusted fair completion estimations, obtained by taking into account the completion times of the other tasks assigned to the same processor, result in better estimates for the fair completion time of a task than the non-adjusted fair completion times. When a task completes execution on a resource the fair share of all the other tasks in the same resource should increase. This is accounted for in the definition of Q_{ij}^{a}, giving a more realistic fair completion time estimate than Q_{ij}. The assignment of tasks to resources with the adjusted mehd is more complex than in the non-adjusted case, since it also requires the exact knowledge of the task workloads. Therefore, we expect the adjusted method to outperform the non-adjusted one, in terms of performance and fairness metrics.

2.2.3 Performance Results

2.2.3.1 *Simulation Environment*

In order to evaluate the performance of the proposed FCTE and adjusted FCTE algorithms we used the GridSim simulator [30] with appropriate extensions and modifications. The Grid Network simulated consists of 3 users, 7 resources and 1 centralized meta-scheduler. Every resource uses the space-sharing policy and consists of 1 machine with 30 CPUs, each of computational capacity equal to 10000 MIPS. The two-phase procedure ("task-ordering" phase, "resource-assignment" phase) is executed by the scheduler every 5 seconds. Each user creates 2000 tasks with the characteristics presented in Table 2. All the tasks have non-critical deadlines, in the sense that tasks whose deadlines expire, continue until their execution is completed (but are recorded as a deadline miss).

Characteristic	Distribution	Mean	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 Inter-arrival Time	Exponential	1/12, 1/20, 1/25, 1/33, 1/40, 1/50, 1/55, 1/60, 1/65, 1/70 secs/task	-	-
Task Deadline	Uniform	-	10 sec	20 sec

Table 2: Task characteristics used in the simulations.

In our experiments we evaluated the performance of the FCTE and the adjusted FCTE scheduling algorithms and compared it to that of other two-phase algorithms proposed in the literature (Table 3).

task-ordering phase	resource-assignment phase
First Come First Servered (FCFS)	Earliest Completion Time (ECT)
Earliest Deadline First (EDF)	Earliest Completion Time (ECT)
Least Length First (LLF)	Earliest Completion Time (ECT)
First Come First Servered (FCFS)	Adjusted Fair Completion Time Estimation (Adjusted FCTE)
First Come First Servered (FCFS)	Non-Adjusted Fair Completion Estimation (FCTE)

Table 3: Simulated algorithms.

2.2.3.2 *Simulation Metrics*

In our experiments we used the following metrics to evaluate the performance of the algorithms proposed:

- Average Excess Time: The average time by which a task misses its non-critical deadline. A tasks *i* excess time is defined as

$$\left| D_i^c - D_i \right|, \tag{17}$$

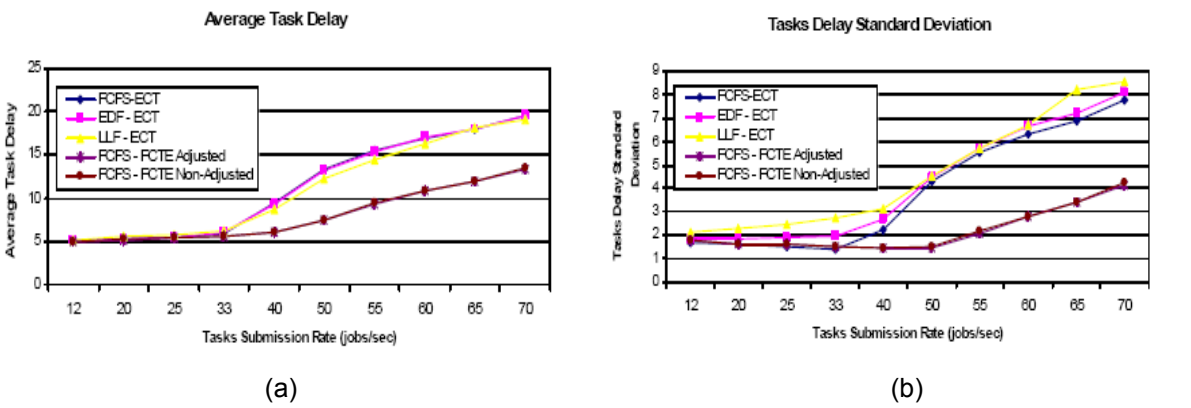
  where D_i^c is the actual completion time of the task *i* and D_i is its non-critical deadline.
- Excess Time Standard Deviation: The standard deviation of the time by which the tasks miss their non-critical deadline.
- Average Task Delay: The average delay of the tasks. A task's delay is defined as the time between the task's creation and the time the results of its execution return to the user.
- Task Delay Standard Deviation: The standard deviation of the task delays.
- Deadlines Missed: The number of tasks that missed their non-critical deadlines.
- Deadline Fairness Metric: The average relative deviation of the demanded task deadlines to the actual task completion times as defined in Eq. (13).

2.2.3.3 *Simulation Results*

In this section we present the performance results obtained from our simulations. As we will see, the FCTE algorithm and the adjusted FCTE algorithm outperform the other scheduling algorithms examined with respect to all the measured metrics (including the fairness metric), especially at heavy load (namely, when the task submission rate increases beyond 40 jobs/sec). The FCTE and the adjusted FCTE algorithms will be found to have very similar performance, indicating that no noticeable benefits can be obtained by using the (more difficult to calculate) adjusted fair completion time estimates instead of the non-adjusted fair completion time estimates.

In Figure 16(a) we illustrate the average task delay as a function of the task submission rate. Generally, the average task delay of all the scheduling algorithms increases when the task submission rate increases. For light load, all scheduling algorithms have similar average delay performance, but when the task submission rate increases beyond 40 jobs/sec the FCTE and the adjusted FCTE algorithms achieve smaller average delay than that of the other algorithms considered. Interestingly, however, when the task submission rate increases beyond 55 jobs/sec the difference between the average delay achieved by the FCTE and the adjusted FCTE algorithms with that achieved by the other scheduling algorithms remains approximtely the same.

Regarding the standard deviation of the task delays, illustrated in Figure 16(b), it is clear that the FCTE and the adjusted FCTE algorithms give smaller delay variance than the other scheduling algorithms considered. This is an indication that the proposed algorithms treat the different tasks in a more fair way than the other scheduling algorithms investigated.



(a) (b)

Figure 16 – (a) The Average Task Delay. (b) The Tasks Delay Standard Deviation.

Figure 17(a) illustrates the average excess time, that is, the average time by which a task misses its non-critical deadline, as a function of the task submission rate. As expected, this metric increases as task submission rate increases. However, the increase is smaller when using the FCTE and the adjusted FCTE algorithms than when using the other algorithms examined. This indicates that when tasks are scheduled using the proposed FCTE algorithms, the time by which they miss their deadlines tends to be smaller than when the other algorithms are used. The standard deviation of the average excess time is presented in Figure 17(b). The standard deviation for the FCTE algorithms is very small, indicating that even tasks that miss their deadlines are treated rather fairly.

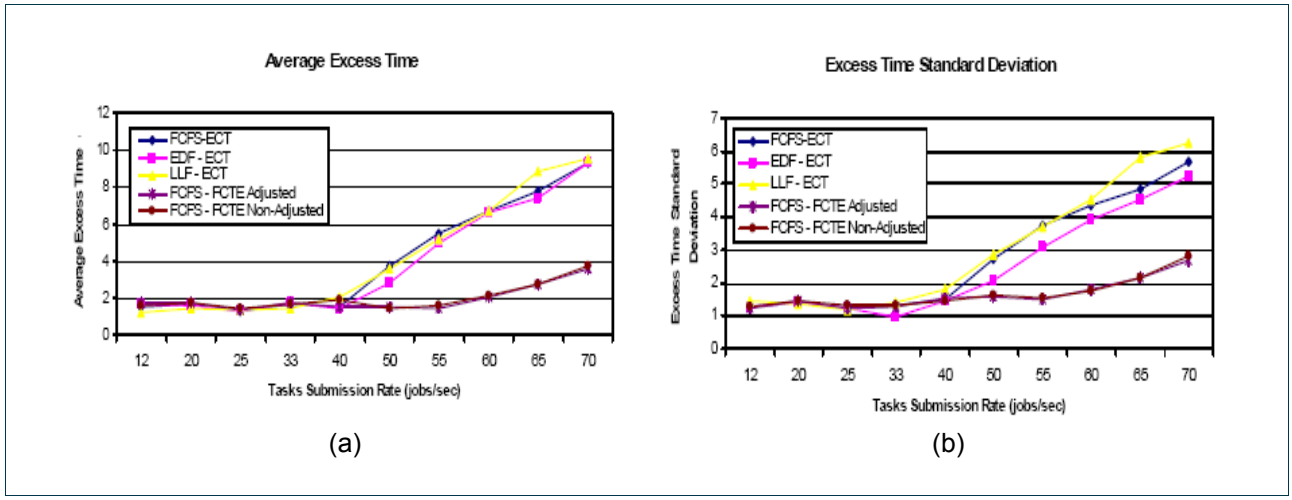


Figure 17 – (a) The Average Excess Time. (b) The Excess Time Standard Deviation.

Figure 18(a) illustrates the number of tasks that miss their non-critical deadline as a function of the task submission rate. As the task submission rate increases, the number of tasks that miss their deadline also increases, for all the scheduling algorithms tested. This is expected, since as the number of tasks submitted increase, it becomes more difficult for the scheduler to meet the tasks deadlines. From the Figure 18 we observe that fewer tasks miss their deadlines, when they are scheduled with the FCTE algorithms than when they are scheduled with other algorithms, and the difference is more significant for large task submission rates.

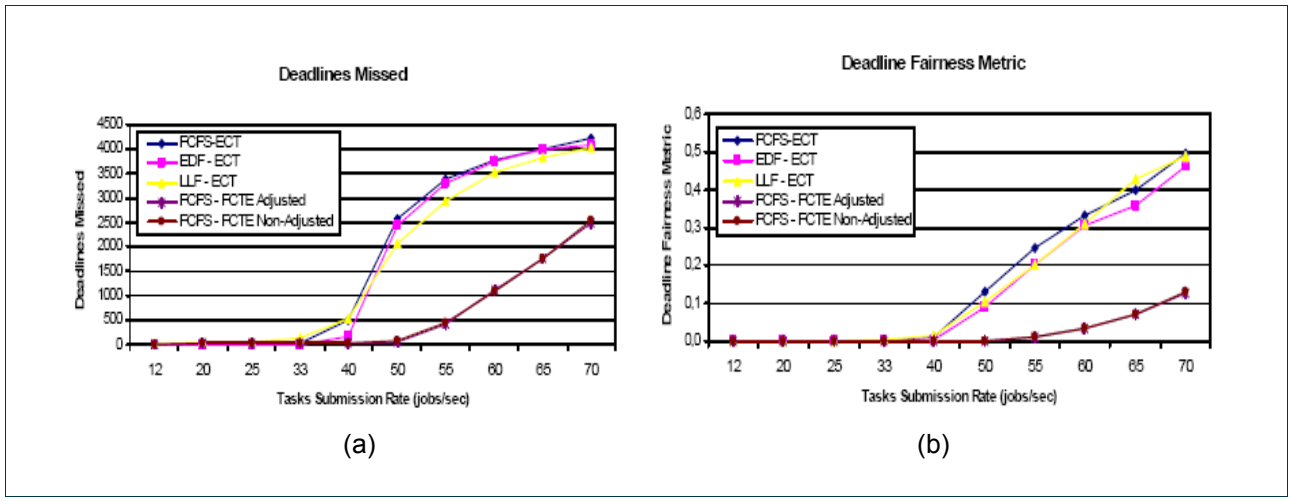


Figure 18 – (a) The Deadlines Missed. (b) The Deadline Fairness Metric.

Figure 18(b) illustrates the deadline fairness metric, defined in Section 2.2.3.2, which measures the relative deviation of the demanded task deadlines to the actual task completion times, as a function of the task submission rate. The results again indicate that fairness, among tasks, is achieved when the FCTE algorithms are used.

2.3 User Fair Scheduling Algorithm

In this section we present a user fair scheduling algorithm for Grid Networks [82]. 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 Best Effort (BE) users 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 users. Also, different weights can be given to users, in which case we talk about weighted user fairness. This could be the case in a university campus, where all the users-students using the Grid infrastructure deserve the same kind of service from it. Our proposed scheme for provides fairness on a per user basis. It draws on corresponding schemes and concepts developed for Data Networks and specifically on Weighted Fair Queuing (WFQ) scheduling algorithm (Appendix D).

2.3.1 WFQ/EST Scheduling Algorithm

Most Grid scheduling algorithms proposed to date schedule BE tasks based on various task characteristics, such as its deadline, workload, estimated completion time, or price the user is willing to pay for its execution. Fairness, however, is another very important criterion that should be taken into account in Grid scheduling. The number of different resource types comprising a Grid Network (computation, communication, storage) makes the application of fairness in Grids a more complex issue than, for example, in Data Networks. In our view, a fair Grid scheduling algorithm should have the following characteristics:

- User Fairness: Fairness should not be enforced on a per task basis, but on a per user basis.
- Joint Fairness: Grids consist of computational, communicational and storage resources. As a result, fairness must be achieved jointly for all these types of resources.
- Task Fairness: Fair scheduling only on a per user basis may not be the best policy, because of different task characteristics and requirements (e.g., deadline, task workload, completion time etc), which should also be incorporated in a fair scheduling algorithm.
- Resource Uniformity: Future Grids will consist of consumers (users willing to pay for the use of the Grid resources) and providers (users offering their resources). In such an environment it is desirable that the resources are fairly used, even when there is a plethora of resources available for the execution of the tasks.

In this section we concentrate on user fairness and 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 it is executed at periodic intervals (Figure 19). 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 [17] (Appendix D). 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.

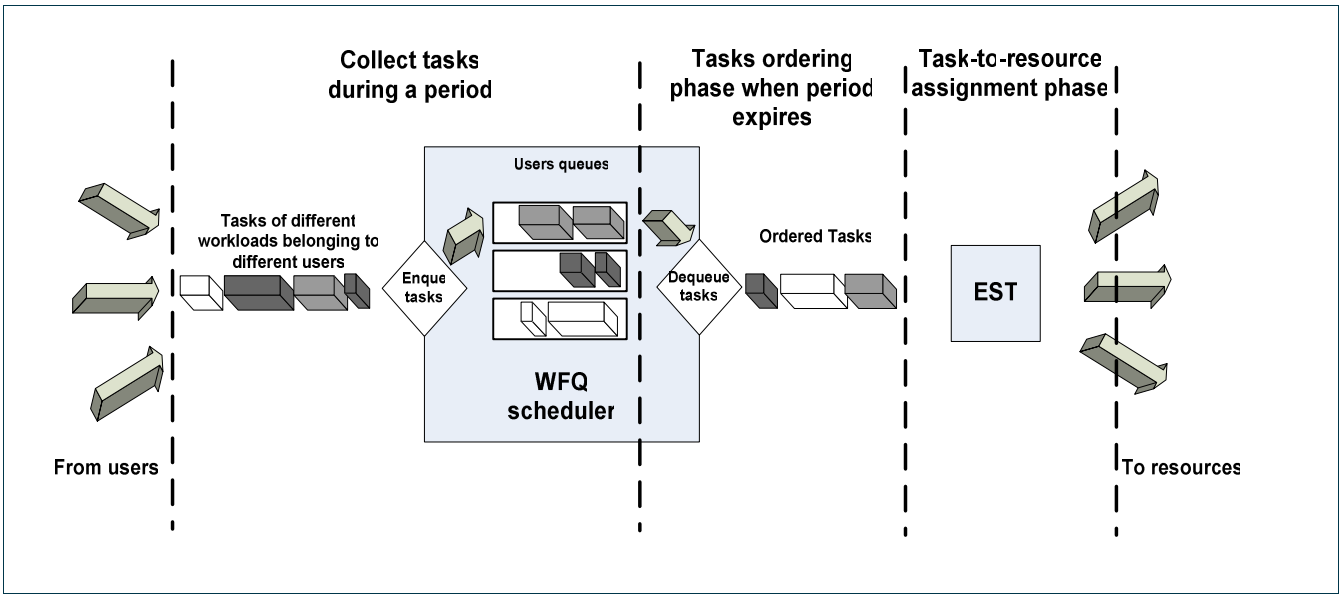


Figure 19 – User fair scheduling using a WFQ scheduler.

The pseudocode of the WFQ/EST scheduling algorithm is presented in Figure 20.

```
1  while period hasn't expired do
2          Collect tasks, by enqueueing them in the WFQ scheduler.
3  end while
4  if period has expired then
5          Ordering Phase: Dequeue tasks from the WFQ scheduler.
6          Assignment Phase: Assign tasks to resources based on EST.
7  end if
```

Figure 20 – WFQ/EST user fair scheduling algorithm.

2.3.2 Performance Results

A number of simulations were conducted in order to validate that the proposed algorithm provides fairness to the users. In our simulations we used 5 users (Table 4), 3 single-machine and single-CPU resources (Table 5), 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 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. 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.

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

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

Resource	Total Computational Capacity
R1	780000 MIPS
R2	520000 MIPS
R3	260000 MIPS

Table 5: Resources computational capacities.

The meta-scheduler uses a two-phase ("task-ordering" and "resource-assignment" phase) procedure for scheduling users, which is executed every 1 second. 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. In our simulations we used the following metrics:

- 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 computation unit delay.
- the deadline fairness metric, is the average relative deviation of the demanded task deadlines from the actual task delays, defined in Eq. (13).

Figure 21 shows the average delay of a computation unit (MI). This metric is used in order to compare the service (per user) the BE users are receiving from the Grid, considering that they generate tasks of different workloads, which are then executed in resources of different computational capacities. We observe that WFQ/EST scheduling algorithm achieves smaller average delay of a computation unit than the EDF/EST scheduling algorithm. This means that the average tasks delay is also decreased and tasks are executed on average faster.

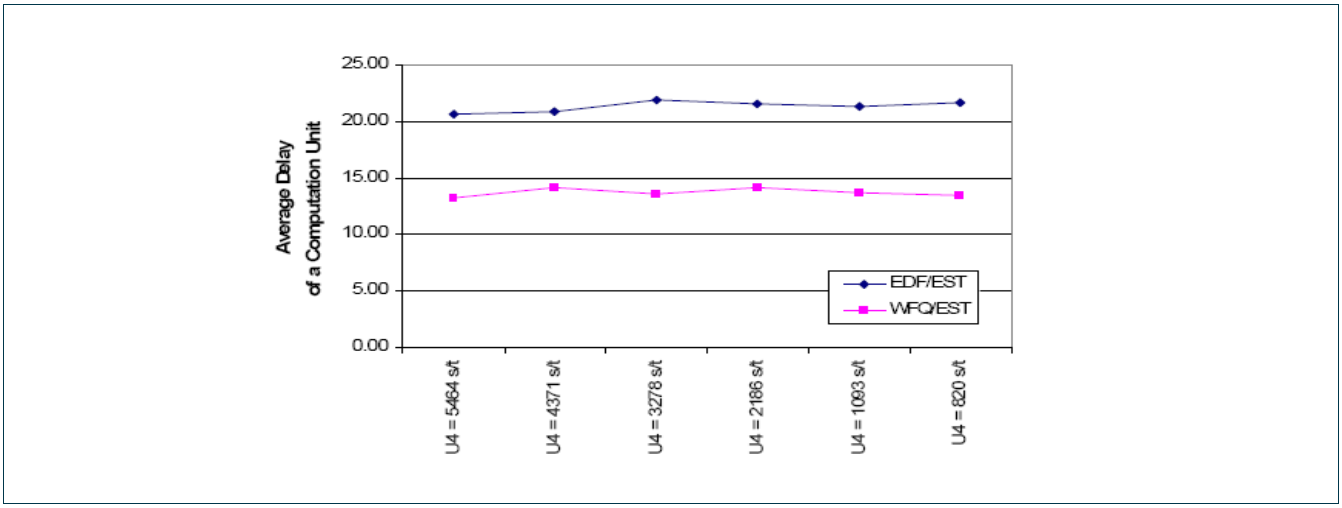


Figure 21 – The average delay of a computation unit, for various task inter-arrival times (secs/task) of BE user U4.

In Figure 22 we observe that the standard deviation of the users computation unit average delay is smaller when using the WFQ/EST user fair scheduling algorithm, than when the EDF/EST scheduling algorithm is used. This is our main user fairness metric and shows that the Grid offers its computational capacity more fairly among different users, when the WFQ/EST algorithm is used. Furthermore, in Figure 23 we observe that the WFQ/EST scheduling algorithm has better deadline fairness than the EDF/EST scheduling algorithm. Also this difference is grown as the user's U4 BE tasks inter-arrival times are decreasing, and more BE tasks (of all the users) are missing their deadlines. So when more tasks are produced than what the Grid Network can serve, then the WFQ/EST scheduling algorithm succeeds in splitting the deadline penalty more fairly among the BE tasks.

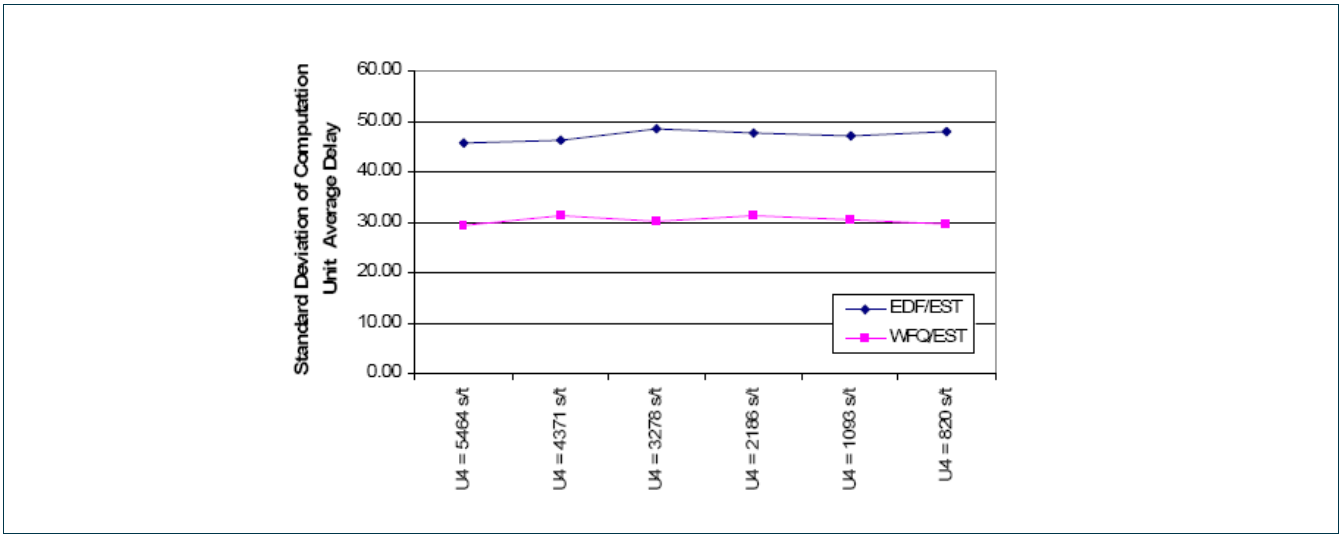


Figure 22 – The standard deviation of computation unit average delay, for various task inter-arrival times (secs/task) of BE user U4.

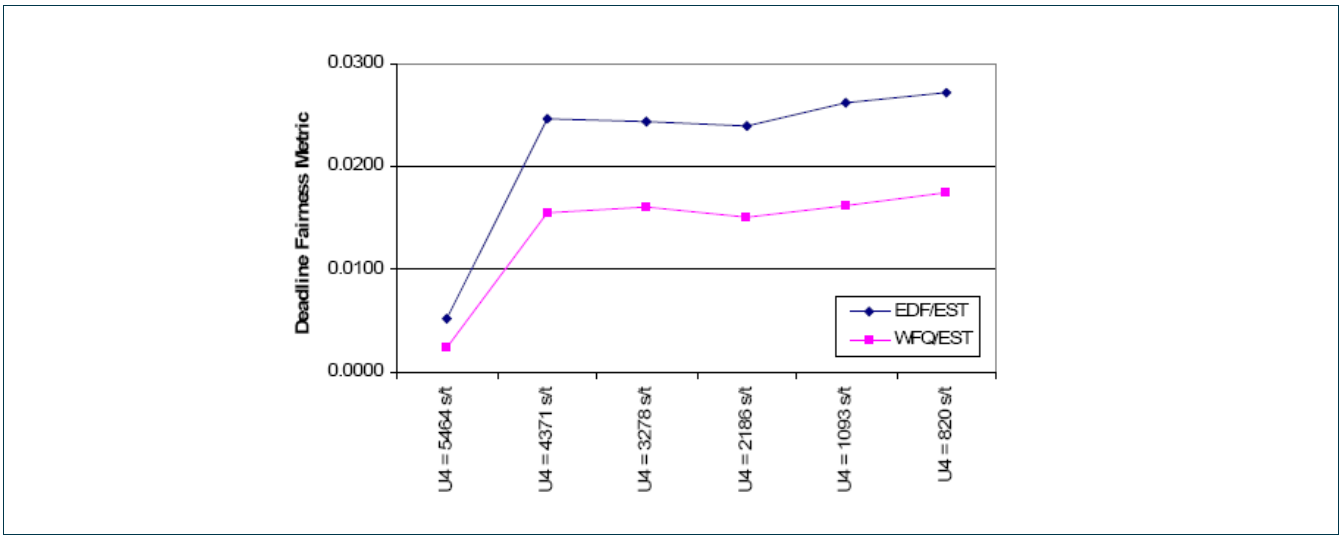


Figure 23 – The deadline fairness metric, for various task inter-arrival times (secs/task) of BE user U4.

2.4 Conclusions

In this section we presented a number of fair scheduling algorithms for Grid Networks. The scheduling algorithms proposed are centralized, “offline” and consist of two-phases: the “task-ordering” phase and the “resource-assignment” phase.

First we presented the Simple Fair Task Order (SFTO), Adjusted Fair Task Order (AFTO) and Max-Min Fair Scheduling (MMFS) algorithms that follow the Max-Min fair sharing scheme. The SFTO and AFTO algorithms provide fairness in the first phase (“task-ordering”), while the MMFS incorporates farness in both phases. The experimental results and the comparisons with the traditional First Come Fist Serve (FCFS) and Earliest Deadline First (EDF) scheduling schemes, indicate that our proposed scheduling algorithms are more fair and better exploit the available Grid resources. The simulation experiments involved the submission of thousands of tasks, of varying length and workload variance to a multiprocessors computing system comprising of hundreds of processors of varying capacity. The results indicate that the MMFS algorithm is less sensitive to processor capacity variations than the SFTO and AFTO schemes. Unde all conditions examined, our proposed algorithms outperformed previously proposed greedy algorithms. We also found that the MMFS scheme outperforms the SFTO and the AFTO schemes in all the simulated conditions.

We also presented the Adjusted and non-Adjusted Fair Completion Time Estimation (FCTE) algorithms that estimate the fair completion time of the tasks, and use this value to assign tasks to processors. In the simulations conducted we observed that the FCTE algorithms outperform other known scheduling algorithms (e.g. the Earliest Completion Time - ECT), with respect to efficiency (average task delay) and fairness related metrics. Also these algorithms provide better Quality of Service (QoS), as is evident from the small number of deadlines missed, and the small average task delay.

Finally, we proposed the WFQ/EST scheduling algorithm that provides fairness on a per user basis instead of on a per task basis, in contrast to the previously described fair scheduling algorithms. This algorithm incorporates fairness in the first (“task-ordering”) phase. The experimental results showed that the proposed algorithm not only provides fairness among users, but also improves performance compared to other tradditional scheduling algorithms.

2.5 Symbols

Symbol	Meaning
T_i	Task *i*
w_i	Workload of task *i*
N	Number of tasks that are scheduled

C	Total computation capacity of the Grid
M	Number of machines/processors in the Grid
c_j	Computation capacity of machine/processor *j*
w_{ij}	Execution time of task *i* on machine/processor *j*
d_{ij}	Communication delay between user *i* and machine/processor *j*
D_i	Deadline of task *i*
γ_j	Estimated completion time of the tasks already running on or already scheduled on machine/processor *j*
δ_{ij}	Earliest starting time of task *i* on machine/processor *j*
δ_i	Grid access delay for task *i*
X_i	Demanded computation rate of task *i*
ρ	Grid normalized load
Ξ	Time interval
φ_i	Integer weight of task *i*
t_i	Non-adjusted fair completion time of task *i*
t_i^a	Adjusted fair completion times of task *i*
r_i	Fair computation rate of task *i*
r_i^s	Schedulable (actual) rate of task *i*
$F_i(t)$	Finish number of task *i* at time *t*
τ	The last time a change in the number of active tasks occurred

$R(\tau)$	Round number at time *τ*
P_j	Set of tasks containing all tasks scheduled on machine/processor *j*
E	The error of the scheduler
E1, E2, E3	Metrics for evaluating fair scheduling algorithms
X_i^c	Allocated computation rate to task *i*
D_i^c	Actual completion time of the task *i*
Q_{ij}	Non-adjusted fair execution time estimation of task *i* in resource *j*
Q_{ij}^a	Adjusted fair execution time estimation of task i in resource *j*
Q_i	Non-adjusted fair completion time of task *i*
Q_i^a	Adjusted fair completion time of task *i*
N_j	Number of tasks assigned to resource *j*

Table 6: Symbols used in Section 2.

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 such resource (time-sharing), are reserved by a user. In Data Networks, QoS mainly refers to packet delay, delay jitter, bandwidth and packet-loss rate. In the case of Storage Area Networks (SAN), QoS can refer to the bandwidth used for data transfers and to the storage capacity (size) available for a task. 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 exact QoS parameters requested (delay, required resources, etc). Then the network, through a procedure called admission control, checks whether it can satisfy the user's request for guaranteed service, without violating the guarantees given previously to other users. If this is possible, then 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 where presented. The first is the General-purpose Architecture for Reservation and Allocation (GARA) [11] and Grid QoS Management (G-QoSM) [15]. These works propose QoS frameworks for the Grid Networks considering the network, the computational and the storage resources. Various other works have concentrated on specific aspects of QoS in Grids, such as network QoS for Grid applications [12], admission control [15] and other. The GARA is perhaps the best known and certainly the oldest framework for supporting QoS in Grid Networks. This framework provides guarantees to an application requesting specific end-to-end QoS characteristics. On the other hand G-QoSM is a newer QoS framework for Grids, which is more actively developed, following the recent trends in Grid Networks. So G-QoSM is Open Grid Service Architecture (OGSA) [13] enabled. Furthermore, G-QoSM incorporates various useful features, such as monitoring and adaptation during an active QoS session. GARA and G-QoSM frameworks reserve for a time period computational resources quantitatively, either by reserving a number of CPUs in a resource or by reserving a percentage of a CPU's capacity (Dynamic Soft Real-time scheduler - DSRT [14]).

In this section we propose a QoS framework for Grid computing [80][81][82]. 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 both theoretically and experimentally that hard QoS, in terms of delay bound guarantees given to each user, can in fact be provided without using 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, either by reserving a number of CPUs in a resource or by reserving a percentage of a CPU's capacity (Dynamic Soft Real-time scheduler - DSRT [14]). Furthermore, in our QoS framework various other useful features are investigated. We consider single and multi-CPU resources, scheduling without a-priori knowledge of task workload, and task migration. We also propose and evaluate computational resources that serve either 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 and usefulness 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. The objectives that we set for the proposed QoS framework are: a) to provide service guarantees to GS users and b) to ensure fair sharing of the resources among BE users. In order to achieve the first objective, the GS users are leaky bucket constrained, so as to follow a constrained task generation generation pattern that is agreed separately with each resource. On the resources, the arriving tasks are queued in a Weighted Fair Queuing (WFQ) scheduler [17] (Appendix D). 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. However, in contrast to most other fairness related works [24], we aim at providing fairness among users and not among individual tasks.

In the following subsections we describe the distributed mechanisms used to provide service guarantees to GS users and fairness to BE 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. Next, 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 (ρ_{ir}, σ_{ir}) constraints (Figure 24) of the user. The parameter ρ_{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 σ_{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, then the GS user is registered to the resource. From then on, 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.

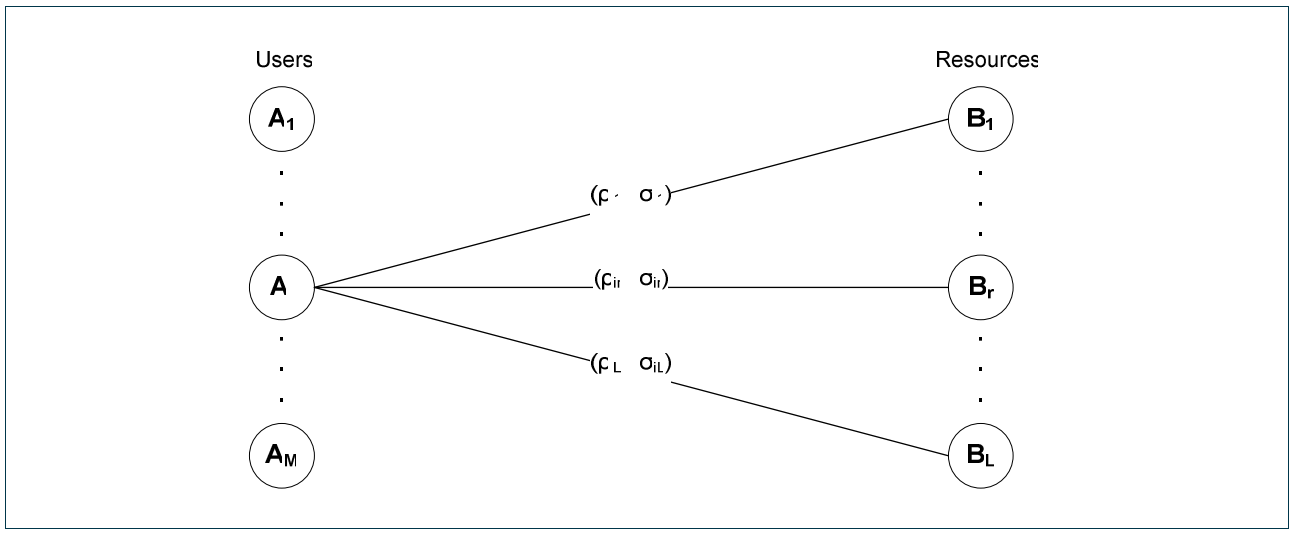


Figure 24 – The (ρ, σ) constrained GS users in the Grid Network.

In order for a resource *r* to accept the registration of GS user *ι*, 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 ρ_{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:

$$\rho_{ir} \le g_{ir}(t) = \frac{C_r \cdot w_{ir}}{\sum_{k=1}^{N_r(t)+1} w_{kr}}, \qquad (18)$$

where C_r is the computing capacity of resource *r*, $N_r(t)$ is the number of GS users already registered to resource *r* at time *t*, and w_{ir} is the weight of the GS user *i* for using the resource *r*. The weights w_{ir} can depend on various parameters, such as the prices the GS users are willing to pay, or their other contributions to the Grid (as in [24]). The Eq. (18) ensures that resource *r* can satisfy the task generation rates of both the new and old GS users.

An additional condition that is agreed during the registration of GS user *i* to resource *r* is that the maximum task workload $J_{ir}^{\max}$ user *i* will ever send to resource *r* will not exceed the resource's maximum acceptable task workload $J_r^{\max}$:

$$J_{ir}^{\max} \le J_r^{\max}. \qquad (19)$$

If both Eq. (18) and Eq. (19) 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.

The pseudocode of the algorithm used for GS user's *i* registration to resource *r* is presented in Figure 25.

```
1   for all resources in the Grid do
2       if resource r satisfies Eq. (18) and Eq. (19) then
3           GS user i registers to resource r.
4       else
5           GS user i does not register to resource r.
6       end if
7   end for
```

Figure 25 – GS user's *i* registration procedure.

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 (ρ_{ir}, σ_{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 (ρ_{ir}, σ_{ir}) controlled with respect to resource *r*, if the following condition is valid:

$$J_{ir}(t) < \sigma_{ir} + \rho_{ir} \cdot t, \ \ \forall t > 0 \, . \tag{20}$$

If a GS task *j* invalidates Eq. (20), then the GS user must locally withhold this task for a time period, denoted by T_{ir}^{j}, until Eq. (20) becomes valid again (Figure 26). So our framework includes in every GS user an admission control (leaky bucket) mechanism, to make sure a task reaches a resource only when some specific constraints are valid.

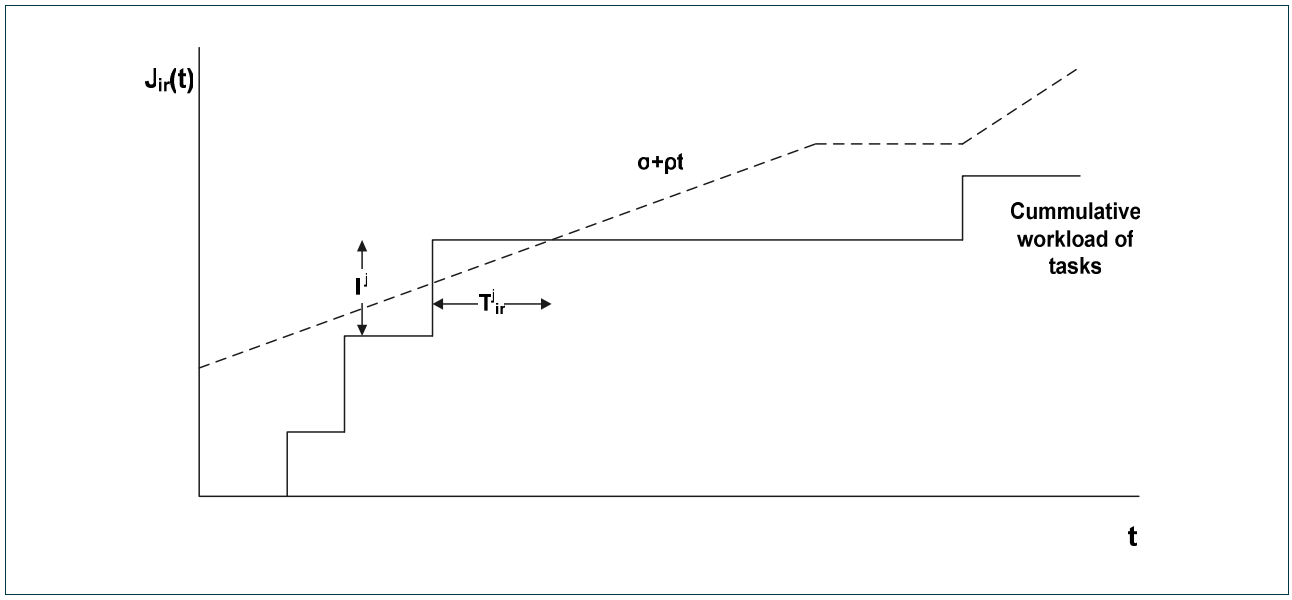


Figure 26 – The GS user is responsible for the observance of his (ρ_{ir}, σ_{ir}) constraints.

When a task is created, the GS user searches for the most suitable resource to which it has already registered. 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* again two conditions must hold. First, the task's workload must not exceed the one agreed,

$$I_i^j \le J_{ir}^{\max} \ , \tag{21}$$

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. If conditions Eq. (18) and Eq. (20) hold and WFQ is used, then it can be proved, by arguing as in [16] (see

Appendix E), that 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

$$\frac{\sigma_{ir}}{g_{ir}} + \frac{J_{ir}^{\max}}{g_{ir}} + \frac{J_{r}^{\max}}{C_r}, \tag{22}$$

where g_{ir} is the minimum value of $g_{ir}(t)$ that does not invalidate Eq. (18) for any registered user. To this delay we must add the total communication delay, denoted by d_{ir}^{j}, required for transferring the task to the selected resource, and the time T_{ir}^{j} the GS user withholds the task in its local queue (Figure 26). So the delay bound B_{ir}^{j} resource *r* guarantees to user *i* satisfies

$$B_{ir}^{j} \le T_{ir}^{j} + d_{ir}^{j} + \frac{\sigma_{ir}}{g_{ir}} + \frac{J_{ir}^{\max}}{g_{ir}} + \frac{J_{r}^{\max}}{C_r}. \tag{23}$$

Based on Eq. (18) and assuming that $w_{ir} = 1$ for all *i, r* we have:

$$B_{ir}^{j} \le T_{ir}^{j} + d_{ir}^{j} + \frac{\sigma_{ir} + J_{ir}^{\max}}{g_{ir}} + \frac{J_{r}^{\max}}{C_r}, \text{ or}$$

$$B_{ir}^{j} \le T_{ir}^{j} + d_{ir}^{j} + \frac{\left(\sigma_{ir} + J_{ir}^{\max}\right)\cdot\left(N_r(t)+1\right) + J_{r}^{\max}}{C_r}. \tag{24}$$

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:

$$B_{ir}^{j} \le \max(T_{ir}^{j}, d_{ir}^{j}) + \frac{\left(\sigma_{ir} + J_{ir}^{\max}\right)\cdot\left(N_r(t)+1\right) + J_{r}^{\max}}{C_r}. \tag{25}$$

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 to a resource *r*, its deadline D_{i}^{j} must be smaller than the resource's delay bound B_{ir}^{j}:

$$D_i^j \le B_{ir}^j . \qquad (26)$$

The pseudocode of the algorithm used in order for a GS user *i* to schedule a task *j* is presented in Figure 27.

```
1   if GS user i is not registered to any resource then
2           Consider task j as a BE task and handle it accordingly.
3   else if GS user i doesn't satisfy  Eq. (20) for any registered resource r  then
4           GS user i withholds the task I, until Eq. (20) becomes valid.
5   else
6           for all GS user's i registered resources, where Eq. (20) holds do
7                   if resource r satisfies Eq. (21) and Eq. (26) then
8                           Schedule task j to resource r.
9                   end if
10          end for
7   end if
```

Figure 27 – GS user's *i* task *j* scheduling.

If more than one resources fulfill the conditions of Eq. (21) and Eq. (26), 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. (25) 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:

$$D(C,J,N,\sigma) \le \max(T,d) + \frac{(\sigma+J)\cdot(N+1)+J}{C}, \qquad (27)$$

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. (25) can be rewritten as:

$$D(C,J,N,m) \le \max(T,d) + \frac{((m+1)\cdot N+1)\cdot J}{C} . \qquad (28)$$

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. (18) 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. (18) 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. (23), becomes

$$B_{ir}^{j} \le T_{ir}^{j} + d_{ir}^{j} + \frac{\sigma_{ir}}{g_{ir}} + \frac{J_{ir}^{\max}}{g_{ir}} + \frac{J_{r}^{\max}}{C_{r}} + R_{r} \qquad (29)$$

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

$$R_{r} \le \frac{J_{r}^{\max}}{C_{r}} \qquad (30)$$

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 7 presents the user/resource combinations that provide delay bounds.

Resource	Delay Bound for GS users
GS	$\max\left(T_{ir}^{j}, d_{ir}^{j}\right) + \frac{\left(\sigma_{ir} + J_{ir}^{\max}\right)\cdot\left(N_r(t)+1\right) + J_r^{\max}}{C_r}$
BE	-
GS_BE_EQUAL	$\max\left(T_{ir}^{j}, d_{ir}^{j}\right) + \frac{\left(\sigma_{ir} + J_{ir}^{\max}\right)\cdot\left(N_r(t)+1\right) + J_r^{\max}}{C_r}$
GS_BE_PRIORITY preemptive	$\max\left(T_{ir}^{j}, d_{ir}^{j}\right) + \frac{\left(\sigma_{ir} + J_{ir}^{\max}\right)\cdot\left(N_r(t)+1\right) + J_r^{\max}}{C_r}$
GS_BE_PRIORITY non-preemptive	$\max\left(T_{ir}^{j}, d_{ir}^{j}\right) + \frac{\left(\sigma_{ir} + J_{ir}^{\max}\right)\cdot\left(N_r(t)+1\right) + 2\cdot J_r^{\max}}{C_r}$

Table 7: Delay bounds given to GS users with respect to the 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 the GS users to the resources is handled by a central meta-scheduler. The meta-scheduler accepts, from the GS users, registration requests containing their requested *(ρ, σ)* parameters. Then the meta-scheduler 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 performed by the GS users are transferred to a central, possibly more powerful, machine. Second, it is possible to use more than one central meta-scheduler in order 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 again a meta-scheduler is responsible for the registration of the GS users to the 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 in this way 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:

$$C'_r = \sum_{j=1}^{M_r} C_{rj} \tag{31}$$

where C_{rj} is the computational capacity of machine *j* and M_r is the total number of machines (CPUs) in resource *r*. However, in the multi-machine resources case the term C_r used in Eq. (18) and in Eq. (23) 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. (18), $g_{ir}(t)$ is the average service rate the 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. (18) has to be replaced by C'_r, yielding

$$\rho_{ir} \le g_{ir}(t) = \frac{w_{ir} \cdot \sum_{j=1}^{M_r} C_{rj}}{\sum_{k=1}^{N_r(t)+1} w_{kr}} \tag{32}$$

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. (23) and in all the other delay bounds given in Section 3.1 has to be replaced by $C_r^{\min}$. For example, Eq. (23) becomes:

$$B_{ir}^j \le T_{ir}^j + d_{ir}^j + \frac{\sigma_{ir}}{g_{ir}} + \frac{J_{ir}^{\max}}{g_{ir}} + \frac{J_r^{\max}}{C_r^{\min}} \tag{33}$$

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. (20) 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 this method is used, the delay bound resource *r* guarantees to user *i* for a task *j* is increased to:

$$T_{ir}^{j} + d_{ir}^{j} + \frac{\sigma_{ir} + n \cdot J_{ir}^{\max}}{g_{ir}} + \frac{J_{ir}^{\max}}{g_{ir}} + \frac{J_{r}^{\max}}{C_{r}}$$

  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. (20) 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 and/or chooses a small *σ*, then an arriving task may have to suffer a long delay *T* waiting for Eq. (20) 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 dynamically, 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. For example, the meta-scheduler may choose to accept the requested *(ρ, σ)* values if the resource has significant computational power, or it may negotiate with the user smaller values if the resource is less powerful.

3.2.5 Framework's Application to Task Migration

Up till now we have assumed the usual scenario where a task is created by a user, then executed on a resource and finally returned back to the user. It is possible, however, under certain conditions, to extend our framework to the case where tasks migrate among a number of resources, before returning the final results to the user. In [26],[27] and [28] various migration policies are presented:

- The user explicitly asks for task migration.
- The resource decides to migrate an executing task to another resource.
- The task itself decides that it needs more or less computational power and decides to migrate to another resource.
- The user specifies the price he is willing to pay for task execution and when a cheaper resource becomes available the task migrates to it.
- The task migrates in response to network failures or DoS attacks, providing in this way fault tolerance.

All these migration policies are dynamic, in the sense that a task starts execution on some resource and may then migrate to some other resource if various runtime conditions hold.

In such cases, where the sequence of resources to be visited is not known a-priori, the proposed QoS framework cannot be applied. It can be applied, however, when the meta-scheduler has a-priori knowledge of the resources a task is going to be executed on (and the maximum task workload to be executed on each resource). This can be the case when data replication strategies are applied and a task needs for its execution various types of data located at different predetermined resources. For example, assume that a task needs for its execution data of type A, B and C, which are stored at different corresponding resources. In this case the task must visit sequentially the three resources, containing the A, B and C type of data. In each resource the task is executed before migrating to the next one, carrying with it any intermediate results. This strategy may be more beneficial in terms of communication times and storage requirements, than transferring all the data (A, B

and *C*) to one resource. So in case it is known a-priori that a task will visit for its execution a static sequence of *K* resources, then Eq. (23) can be expressed as following:

$$B_{ir}^{j} \le T_{ir}^{j} + \sum_{r=1}^{K-1} d_{ir} + \frac{\sigma_{ir}}{g_{ir}} + \sum_{r=1}^{K-1} \frac{J_{ir}^{\max}}{g_{ir}} + \sum_{r=1}^{K-1} \frac{J_{r}^{\max}}{C_{r}} + \sum_{r=1}^{K-1} R_{r} \qquad (34)$$

where T_{ir}^{j} is the maximum input delay of task *j* for all the resources in the sequence.

3.3 Simulation Results

3.3.1 Simulation Environment and Assumptions

We implemented the centralized version of the proposed QoS framework in the GridSim simulator [30]. 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. In our simulations, a GS user uses the same *(ρ, σ)* parameters for all the resources it registers to.

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.

3.3.2 Parameters and Scenarios

In order to obtain realistic simulation parameters, we used the results of the Grid profiling study of [29], 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 results and numeric data we decided to simulate three GS users, named U1, U2 and U3, corresponding to three of the five VOs presented in [29] (the Atlas, Magic and Dteam VOs). Using the VOs average task execution times and processor speed (estimated to be about 26000 MIPS for the processors in the kallisto.hellasgrid.gr cluster) we calculated their corresponding average task workloads, measured in Million Instructions (MI). Based on [29] we decided the following simulation parameters for modeling GS users:

User	Characteristic	Distribution
Atlas/U1	Task Workload	Fixed: 419900000 MI
Magic/U2	Task Workload	Fixed: 71383000 MI
Dteam/U3	Task Workload	Fixed: 419900 MI
Atlas/U1	Task Inter-arrival	Fixed: 546 secs/task (s/t)
Magic/U2	Task Inter-arrival	Fixed: 3278 secs/task (s/t)
Dteam/U3	Task Inter-arrival	Fixed: 6010 secs/task (s/t)

Table 8: GS users task workload and inter-arrival time.

Also, based on Table 8, the *(ρ, σ)* constraints of each GS user were calculated (Table 9). Specifically, the *ρ* parameter of each 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 corresponding GS user's average task workload.

User	Characteristic	Distribution
Atlas/U1	*ρ*	Fixed: 768473 MIPS
Magic/U2	*ρ*	Fixed: 21773 MIPS
Dteam/U3	*ρ*	Fixed: 699 MIPS
Atlas/U1	*σ*	Fixed: 2099500000 MI
Magic/U2	*σ*	Fixed: 356915000 MI
Dteam/U3	*σ*	Fixed: 2099500 MI

Table 9: GS users *(ρ, σ)* constraints.

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 10). The task workloads submitted by these users were equal to the Atlas/U1 average task workload (namely, 10000 MI).

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

Table 10: BE users task workloads and inter-arrival time.

At the kallisto.hellasgrid.gr cluster presented in [29], 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 (Table 11). The resource type scenarios examined are presented in Table 12. For example, in the GB scenario of Table 12, 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.

Resource	Simulated CPUs	Total Computational Capacity
R1	30	$30 \cdot 26000 \approx 780000$ MIPS
R2	20	$20 \cdot 26000 \approx 520000$ MIPS
R3	10	$10 \cdot 26000 \approx 260000$ MIPS

Table 11: Resources computational capacities.

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. (22) 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 8). 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 12: Resources scenarios.

3.3.3 Performance Metrics

In our simulations we recorded 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.

3.3.4 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 12. The task workloads and inter-arrival times follow a fixed (deterministic) distribution, using the

values of Table 8 and Table 10, 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.

3.3.4.1 *Guaranteed Delay Bounds for GS Users*

Figure 28 shows that our scheme succeeds in providing hard delay guarantees to the GS users. Figure 28 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 11) fewer tasks miss their deadlines, but in the GBE resource scenario many BE tasks fail (Figure 29). 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.

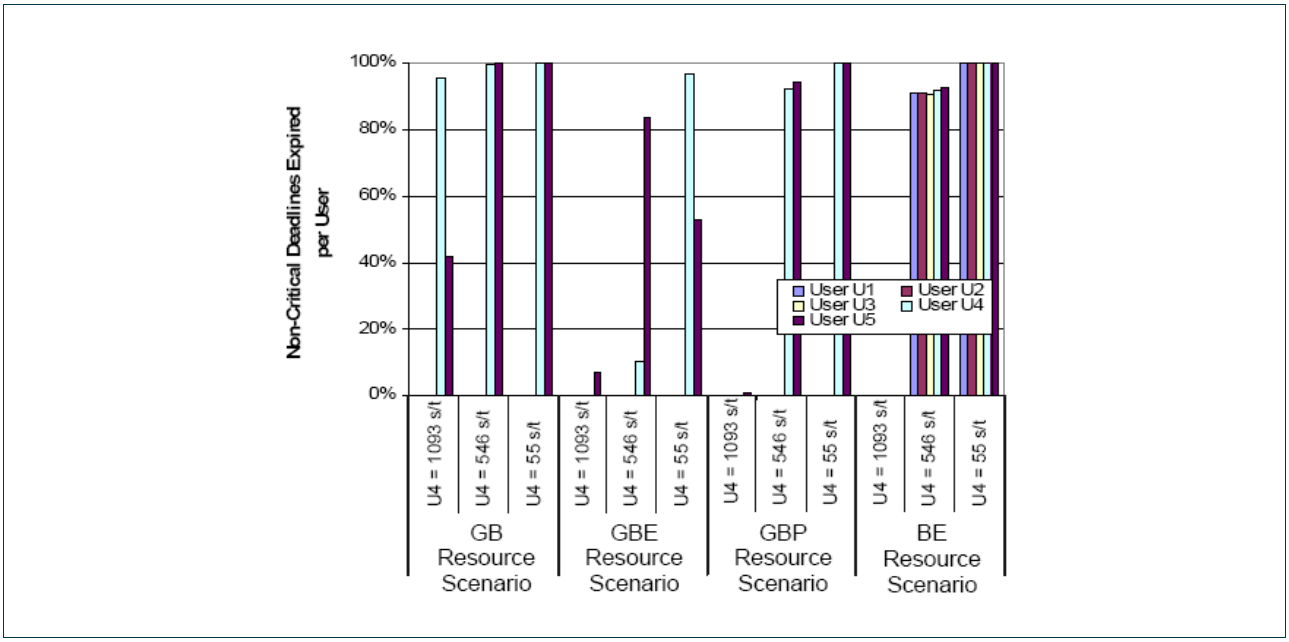


Figure 28 – The per user percentage of the number of tasks that miss their non-critical deadlines, for various resource scenarios and task inter-arrival times (in secs/task) of BE user U4.

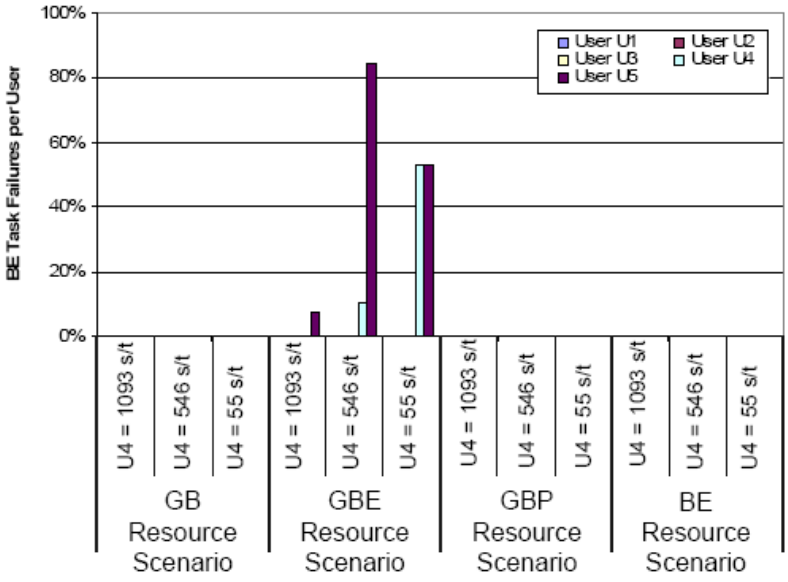


Figure 29 – The per user percentage of the number of failed tasks, for various resource scenarios and task inter-arrival times (in secs/task) of BE user U4.

In Figure 30 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 31 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.

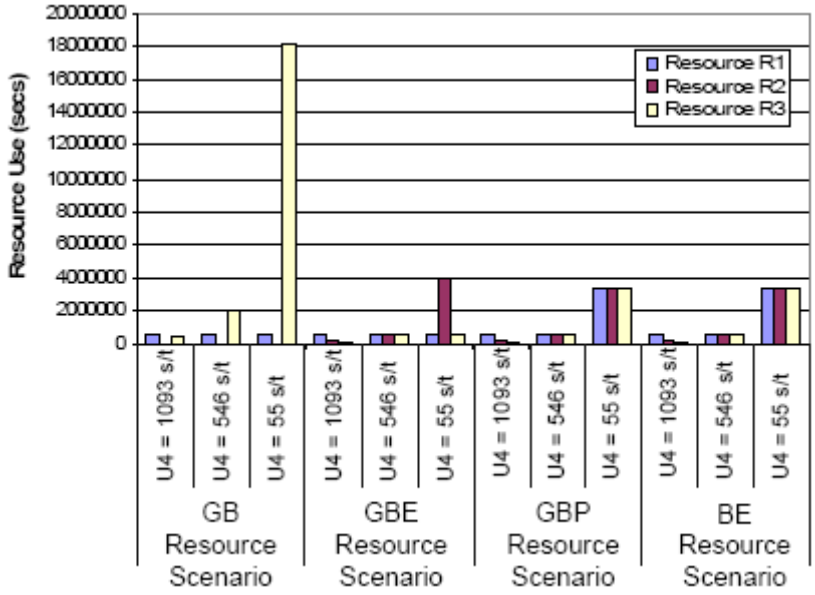


Figure 30 – The resource use, for various resource scenarios and task inter-arrival times of BE user U4.

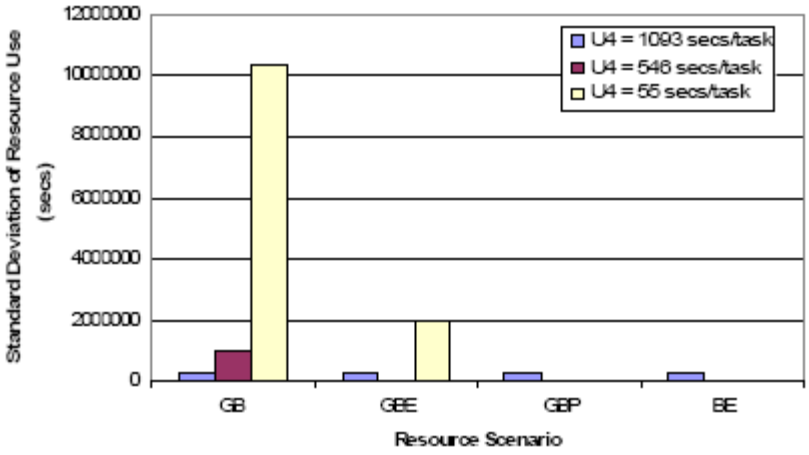


Figure 31 – The standard deviation of the resource use, for various resource scenarios and task inter-arrival times of BE user U4.

3.3.4.2 *(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. (20) 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 8 and maximum value 5%, 20%, or 50% larger than the corresponding fixed value of Table 8, 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 8 and maximum value 20% larger than the corresponding fixed value of Table 8, to be referred as Mixed distribution.

Figure 45 shows the per user percentage of tasks that miss their non-critical deadline in the GBP resource scenario (Table 12), 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 46 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).

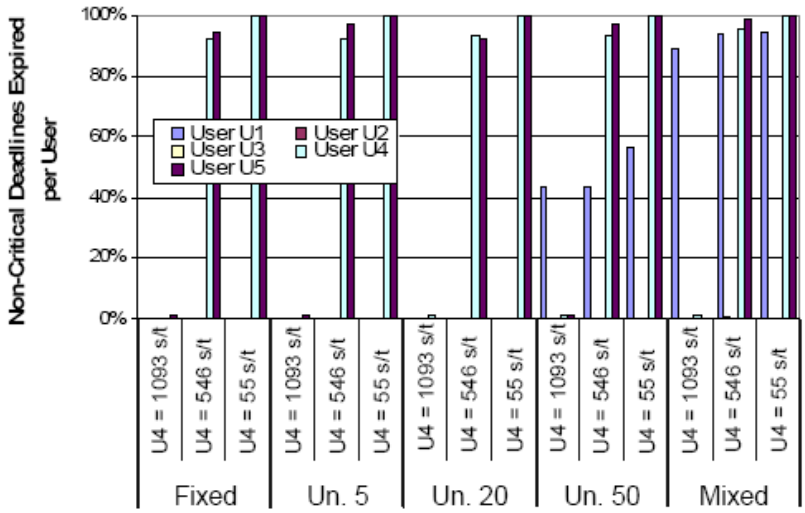


Figure 32 – The per user percentage of the number of tasks that miss their non-critical deadlines using the GBP resource scenario, for various task inter-arrival times (secs/task) of GS users U1, U2, U3 (Fixed, Un.5, Un.20, Un.50, Mixed) and of BE user U4.

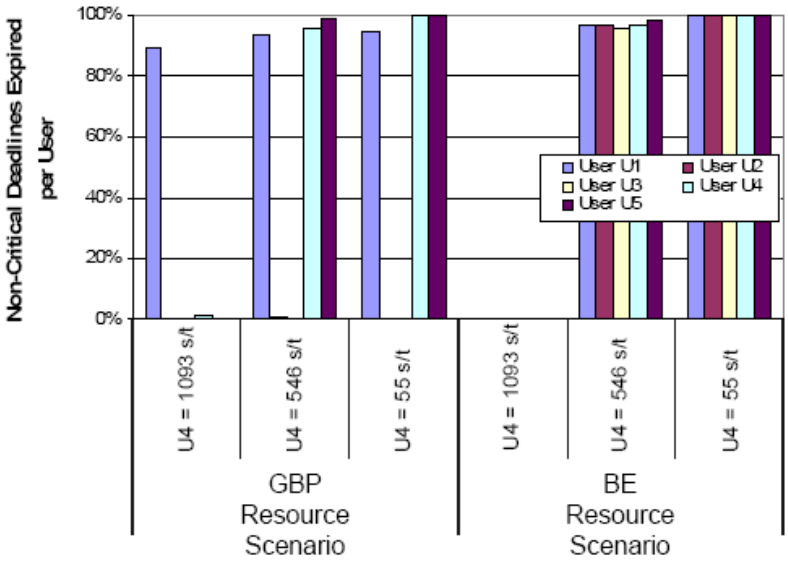


Figure 33 – The per user percentage of the number of tasks that miss their non-critical deadlines, for various resources scenarios and task inter-arrival times (in secs/task) of BE user U4. We assume inter-arrival times that follow the Mixed distribution for the GS users.

3.3.4.3 *Static against Dynamic Registration*

We also conducted a number of experiments to evaluate the benefits of the dynamic against the static registration of the GS users to the resources. As mentioned previously, in our simulations the GS user U1 registers to resource R1, while U2 and U3 to R2. In the experiments conducted for this section, we decreased the number of tasks the GS user U1 generates, this way at a point at time during an experiment U1 stops producing new tasks and so R1 becomes available for use from the other GS users. When static registration is used, U2 and U3 users do not take advantage of resource R1, while when dynamic registration is used U1 unregisters from R1 and U2, U3 register to it.

In our experiments we observe the benefits of the dynamic registration mainly in the GBE resource scenario. At this scenario fewer BE tasks miss their non-critical deadlines (Figure 34), than when static registration is used (Figure 35), because the BE tasks of user U5 can also use the R1 resource, when the U1 unregisters from it. For the same reason fewer tasks of the U5 user fail in the dynamic case, than in the static case.

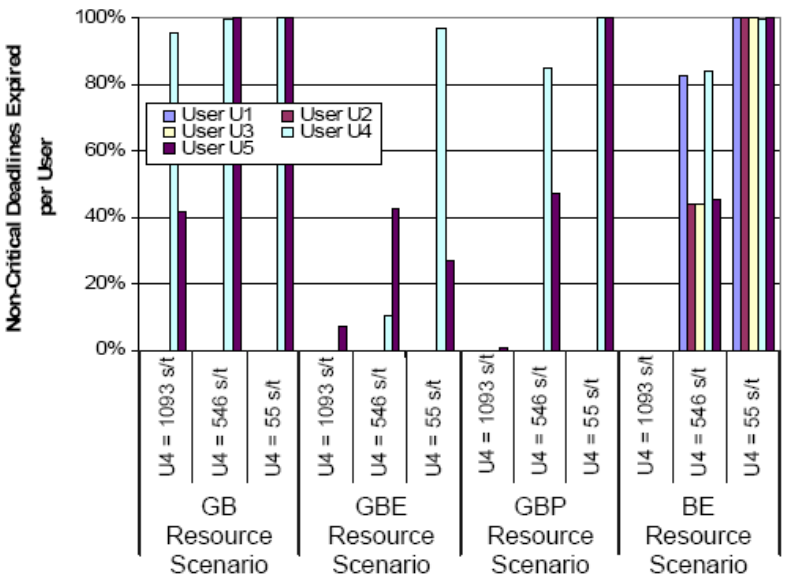


Figure 34 – The per user percentage of the number of tasks that miss their non-critical deadlines using dynamic registration, for various resources scenarios and task inter-arrival times (secs/task) of BE user U4.

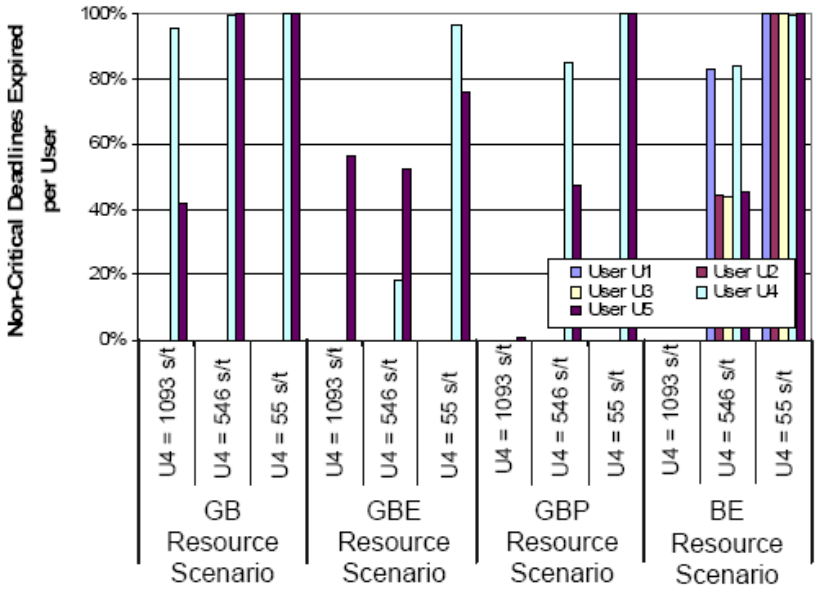


Figure 35 – The per user percentage of the number of tasks that miss their non-critical deadlines using static registration, for various resources scenarios and task inter-arrival times (secs/task) of BE user U4.

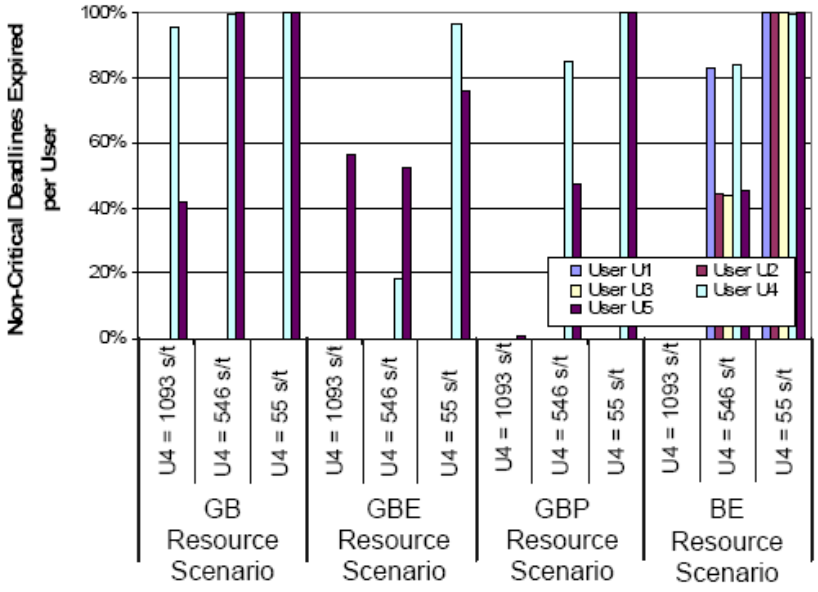


Figure 36 – The per user percentage of the number of tasks that miss their non-critical deadlines using multi-CPU resources, for various resource scenarios and task inter-arrival times (secs/task) of BE user U4.

3.3.4.4 *Multi-CPU Resources*

We conducted a number of experiments using resources with multiple CPUs and measuring the per user percentage of the number of tasks that miss their non-critical deadlines over the total number of tasks each user creates. Figure 36 shows the results obtained for the case where the R1, R2 and R3 resources have 30, 20 and 10 CPUs, respectively. Each CPU had computational capacity equal to 26000 MIPS. In the experiments conducted the delay bounds given to the GS users and the deadlines of all users tasks are calculated based on Eq. (20). From the figure we observe that in all cases the GS users (U1, U2, U3) do not miss their deadlines. So even when multi-CPU resources are used, our framework again succeeds in providing QoS, in terms of hard delay guarantees, to the GS users.

3.3.4.5 *Scheduling Without A-priori Knowledge of Task Workloads*

In this subsection we evaluate the performance of the proposed QoS scheme when task workloads are not known a-priori. The GS user task inter-arrival times and *(ρ, σ)* constraints used are those of Table 8 and Table 9. The GS task workload scenarios used are presented in Table 13, while the BE task inter-arrival times and workloads are those of Table 10. U4 user's task inter-arrival times equal to 1093 secs/task. Finally, in our experiments we focused in the GBP resource scenario, and assumed that the BE task workloads are always known.

Workload Scenario	Distribution
S0	Exponential: Average task workload equal to the orig. value U1 = 419900000 MI U2 = 71383000 MI U3 = 419900 MI
S-10	Exponential: Average task workload equal to -10% of the orig. value U1 = 377910000 MI U2 = 64244700 MI U3 = 377910 MI
S-30	Exponential: Average task workload equal to -30% of the orig. value U1 = 293930000 MI U2 = 49968100 MI U3 = 293930 MI

Table 13: GS users task workloads.

Figure 37 presents the percentage of non-critical deadlines missed by GS users, while Figure 38 presents the percentage of GS tasks that are backlogged at the input of the Grid waiting for Eq. (20) to become valid. The Exact method presented in the figures corresponds to the case where we have exact a-priori knowledge of the

task workloads, in which case we can always accurately detect GS user constraint violations, and either handle the corresponding tasks as BE tasks or backlog them (Figure 38). Some of the tasks handled as BE tasks are the ones missing their deadlines in Figure 37. The Exact method produces the smallest number of tasks missing their deadlines, since it uses the correct task workload values to accurately detect when the *(ρ, σ)* constraints are to be violated.

The Conservative and the Full-Aggressive methods, however, cannot always accurately detect the GS user constraint violations and handle many tasks as GS tasks, instead of BE tasks, even when their deadlines cannot actually be met. This leads to many GS tasks missing their deadlines (Figure 37), while no tasks are ever backlogged at the input (Figure 38). Specifically, in the Conservative method the scheduler assumes that all tasks have the same (maximum) workload, even when the task workloads are actually different (smaller or bigger), while in the Full-Aggressive method the update messages sent to the scheduler about an executed task's actual workload, may arrive too late. As a result the $J_{ir}(t)$ variable of Eq. (20) is not updated correctly or on time.

The Conservative Task Submission with Feedback method, produces better results than the Conservative and the Full-Aggressive methods, in terms of deadlines missed (Figure 37), because GS user violations are often detected and tasks are backlogged (Figure 38). Again, many GS tasks miss their deadlines, but some of them are handled as BE tasks.

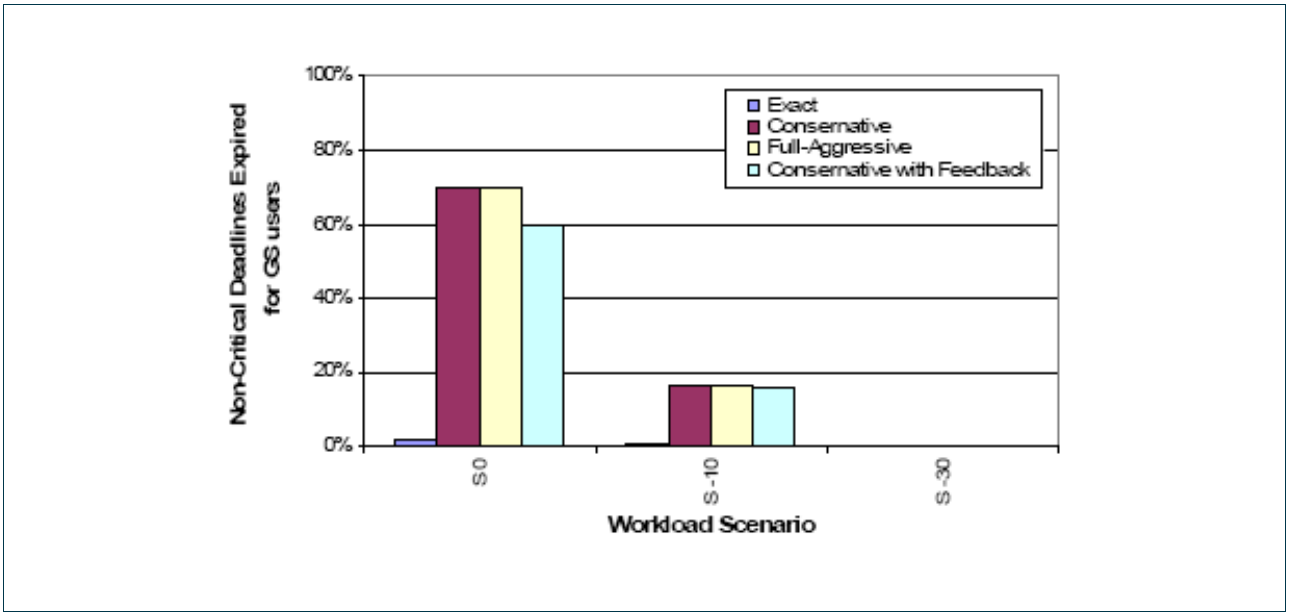


Figure 37 – The percentage of the non-critical deadlines-missed by GS users using the GBP resource scenario, for various workload scenarios and task inter-arrival times for BE user U4 equal to 1093 secs/task.

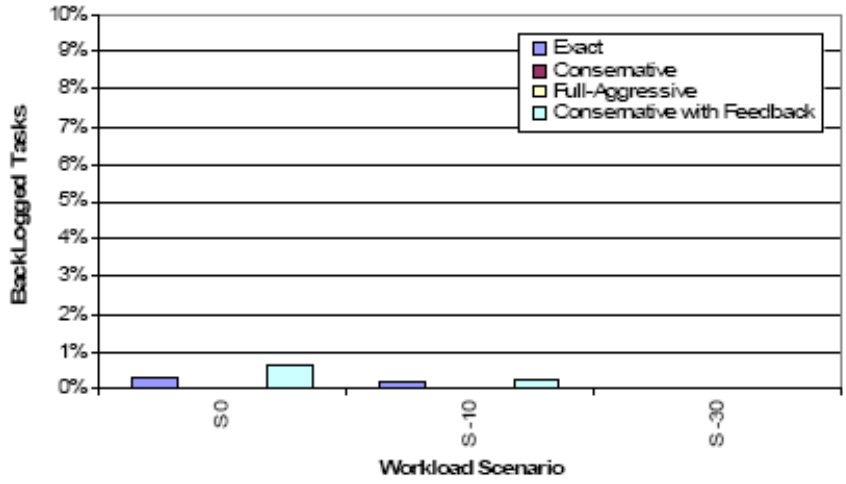


Figure 38 – The percentage of GS tasks which are backlogged using the GBP resource scenario, for various workload scenarios and when the task inter-arrival time of BE user U4 equals to 1093 secs/task.

3.4 Conclusions

We proposed and theoretically and experimentally analyzed a Quality of Service (QoS) framework for Grid Computing that provides hard delay guarantees to Guaranteed Service (GS) users and best effort service to Best Effort (BE) users. Our simulation study indicates that the proposed 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. Finally, various other useful features were investigated in the context of our QoS framework, such as scheduling without a-priori knowledge of task workloads and task migration.

3.5 Symbols

Symbol	Meaning
ρ_{ir}	The long term task generation rate
σ_{ir}	The maximum workload of tasks (burstiness) that the GS user will ever send

C_r	The computing capacity of resource *r*
$N_r(t)$	The number of GS users already registered to the resource *r* at time *t*
w_{ir}	The weight of the GS user *i* for in the resource *r*
$J_{ir}^{\max}$	The maximum task workload the GS user will ever send to the specific resource
$J_r^{\max}$	The resource's maximum acceptable task workload
T_{ij}	The time period for which the GS user i must locally withhold a task j, in order to preserve his (ρ, σ) constraints
D_i^j	User's i task *j* deadline
I_i^j	User's i task *j* workload
d_{ir}	Total communication delay, between user *i* and resource *r*.
C'_r	The total computational capacity of a multi-CPU resource's r
C_{rj}	The computational capacity of the CPU *j,* of the resource *r*
M	The total number of CPUs in the resource r
$C_r^{\min}$	The lowest computational capacity of a CPU, in the resource r

Table 14: Symbols used in Section 3.

4 Co-allocation of Grid Resources

This section addresses the problem of co-allocation of Grid resources. We assume that a user (an application or a Virtual Organization - VO) submits a task on the Grid Network and requests a number (more than one and possibly of a different kind) of service guarantees. The submitted task can consist of a number of other "subtasks" with various interdependencies. Thus, the user requests a co-allocation of (possibly different) resources and service guarantees for the "execution" of the "subtasks" on these resources. We consider two classes of co-allocation: (a) Concurrent co-allocation, in which the guarantees are requested in the same time frame, simultaneously, and (b) Workflow, in which the guarantees are requested in different time frames. Workflows generally include a number of different steps that have to be executed on different resources.

Resource co-allocation is one of the most challenging problems in Grids. The co-allocation problem for computational Grids has been defined in [33]. In the Condor project, the gang matchmaking scheme [31] extends the matchmaking model in order to support the co-allocation of resources. [32] addresses the potential benefit of sharing jobs between independent sites and/or using multi-site applications. In the Koala Grid scheduler [36] jobs are composed of components. Components are primitive sub-jobs ("subtasks") that are expected to run concurrently, at the same time period and for the same duration. Koala monitors resource availability in all of the Grid organizations. When enough resources are available, it selects some jobs from its queues for execution.

In order to co-allocate resources as defined in the Workflows class, the scheduler has to orchestrate resources belonging to different sites and to different administrative domains. To do so advance reservation of these resources has to be supported by the local resource management systems. For more information about advance reservation of resources please refer to D5.4 of the Phosphorus project [78]. 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 [34] has been proposed by the GRAAP working group in the Open Grid Forum (OGF). This approach can be used as a simple negotiation protocol.

Some types of joint communication and computation problems have also been examined. In [44] the authors decoupled the data replication and computation problems and evaluated the performance of data and task schedulers working in a cooperatively manner. In [45] 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 [46]. The authors in [47] introduced the concept of scheduling and routing of advance reservation requests in the communication plane. More specifically, in [47] 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. A framework for specifying and handling co-reservations in Grid environment is presented in [35]. This framework can be applied to the reservation of applications running concurrently on multiple resources and to the planning of job flows, where the components may be linked by some temporal or spatial relationship.

Finally, a taxonomy for workflow management systems in Grid Computing is presented in [37]. Grid workflow management systems with scheduling algorithms have been developed by several projects. Condor DAGman [38] accepts DAG (Direct Acyclic Graphs) description files representing workflows. Next based on the order of the tasks and the dependency constrains in the description files, Condor DAGman submits tasks to Condor-G. Condor-G schedules these tasks onto the best machines available, using a FIFO strategy without any long-term optimization. Pegasus [39] handles abstract workflows which are composed of tasks and their dependencies and concrete workflows which are the mappings of abstract workflows to Grid resources. To serve these requests, Pegasus searches available application components which produce the required data products and available input and intermediate data replicas in the Grid. In GridFlow [40], workflow scheduling is conducted hierarchically by a global Grid workflow manager and a local Grid sub-workflow scheduler. Gridbus [41] architecture is driven by the requirements of Grid economy. A few more examples of workflow management systems are: ASKALON [42], ICENI [43] and UNICORE [73].

In this section we formulate different problems of co-allocation of Grid resources and propose solutions to these problems.

In Section 4.1 we examine the “Anycast Communication and Computation” problem, present two variations to this problem and two algorithms to address these variations. In the first case we want to concurrent co-allocate the communication and computation resources. In the second case we assume that task processing consists of two successive steps: (i) the transfer of data from a site to the computation resource and (ii) the execution of the task at the computation resource, forming in this way a simple workflow.

In Section 4.2 we address a problem of concurrent co-allocation of network, storage and computation resources. We present an ILP formulation for the scheduler that is executed at periodic instances. Thus, the scheduler determines whether or not a job is accepted for execution, and if so, which resources it may use.

In Section 4.3 we present the MetaScheduling Service (MSS) developed in the VIOLA project. The MSS of VIOLA is based on UNICORE and can tackle complicated workflows allowing the end-user to execute the individual components of his application using the most appropriate resources available. The orchestration of resources of different sites belonging to different administrative domains is done by this MSS. Finally, we turn our attention to the advance reservation of network resources and present a brief overview of the network reservation system ARGON.

4.1 Anycast Routing and Scheduling algorithms

In this section we address the "Anycast" routing and scheduling problem, defined as follows: Given the Grid network we try to find the computation resource to execute a task and the path over which to route the data of the task. We present two variations to this problem and two algorithms to address these variations.

Firstly, we examine the problem of concurrent co-allocation of the communication and computation resources. We present a multi-constrained algorithm that uses the path delay and the computation load as selection metrics. The algorithm is an online, one-phase, centralized algorithm. We then 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 and (ii) the execution of the task at the computation resource. We present a multicost algorithm for the joint scheduling of the communication and computation resources. The proposed algorithm selects the computation resource to execute the task, determines the path to route the input data, finds the starting times for the data transmission and the task's execution. Then, advance reservations on the corresponding communication and computation resources have to be performed. The algorithm is an online, one-phase, distributed algorithm that can be easily extended to work in a centralized manner.

4.1.1 Multiconstrained Scheduling and Routing

This section details the network-based server selection. Section 4.1.1.1 introduces an abstraction of the physical network topology, allowing traditional routing algorithms to cope with anycast destinations. Besides single constraint path length from source to destination, other metrics are also taken into consideration for the target selection. Indeed, selecting the most suitable server to accomplish a specific task might depend on a combination of multiple criteria such as end-to-end delay, path cost or the current server load. This leads to a multi-constrained optimal path (MCOP) problem, for which the exact QoS routing algorithm SAMCRA is considered. A new sub-path evaluation ordering method for this algorithm is presented in Section 4.1.1.2.

4.1.1.1 *Routing towards Anycast Destinations*

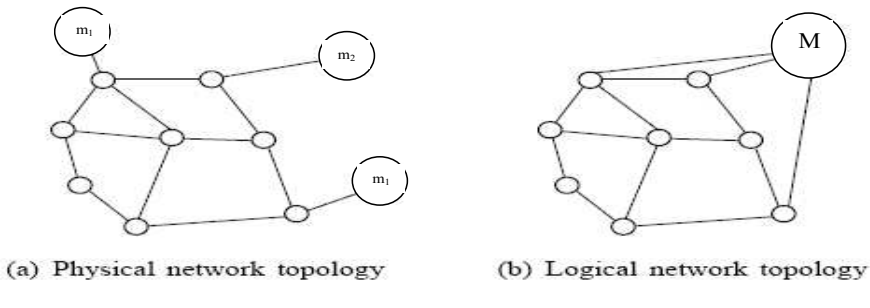


Figure 39 - From a physical perspective, anycast members are scattered all over the network, while they can be considered as a single logical node with multiple connections to the network.

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 computation resources (computation machines). From a routing perspective, all "anycast" nodes $m_j \in M$ can be grouped in a single virtual anycast node *M*, as depicted in Figure 39. This abstraction is only applicable if the target anycast group is a *final routing destination*, because the virtual anycast node cannot forward packets. For the application in mind, anycast nodes are computational resources and not routers, making this a realistic assumption. Furthermore, this approach requires a distinct logical network topology—and hence a distinct routing component instance—for each anycast group present in the routing domain. In general, however, the number of anycast groups in an autonomous system will probably be small because of their focused applicability.

When traditional single constraint shortest path routing is applied to the logical topology, *shortest shortest path routing* as discussed in [60] is achieved. For the remainder of this section, additional metrics are also taken into account, so Dijkstra shortest path routing or other single constraint routing algorithms are not sufficient. In a multi-constrained anycast routing problem, each network link *l* is characterized by *n* link weights $w_i(l) \geq 0$ for all $1 \leq i \leq n$. We will consider only additive weights here, i.e. the cost/weight of a path is the sum of its links' weights (note that multiplicative weights can be converted to additive ones using the log function). Besides additive network-related constraints such as delay or hop count, also server-related constraints can be taken into account (e.g., server load). In this case, edges not directly attached to a member of the anycast server group have a weight equal to zero for all server-related constraints. Therefore, only the last edge of a path *p* from a source node *S* to an anycast member m_j can have a non-zero value for the server-related components of the weight vector. Based on [59], the corresponding anycast MCOP problem can be defined as follows: Given *n* constraints L_i ($1 \leq i \leq n$), find a path *p* from a source node *S* to the virtual anycast node $m \in M$ which satisfies the following condition:

$$w_i(p) \overset{def}{=} \sum_{l \in p} w_i(l) \leq L_i \quad ,1 \leq i \leq n \tag{35}$$

Additionally, for some length function *len*(.), the condition *len*(*p*) ≤ *len*(*p'*) should hold for all paths p and *p'* between *S* and *m*. This anycast multiple constraints routing problem can now be solved by applying the SAMCRA algorithm, which is discussed in the next section.

4.1.1.2 *Exact Multiple Constraints Routing: SAMCRA*

In this section, the SAMCRA algorithm with look-ahead extension is briefly explained in order to present an adaptation of the original SAMCRA sub-path evaluation ordering. For an in-depth investigation of the algorithm, the reader is referred to [62]. SAMCRA is based on four key concepts: non-linear path length, k-shortest paths, non-dominated paths and look-ahead. Before presenting the complete algorithm, these key concepts are clarified. The q-vector norm of path *p* from source *S* to destination *D* can be computed as follows:

$$len_q(p) = \left(\sum_{i=1}^{n} \left(\frac{\sum_{l \in S \to D} w_i(l)}{L_i} \right)^q \right)^{\frac{1}{q}} \quad (36)$$

For q → ∞, Eq. 36 can be rewritten as follows:

$$len_\infty(p) = \max_{i=1,\ldots,n} \left(\frac{\sum_{l \in S \to D} w_i(l)}{L_i} \right) \quad (37)$$

According to P. Van Mieghem et al. [59], finding the shortest path between *S* and *D* using the *non-linear length function* in Eq. 37, solves the MCOP routing problem. The *k-shortest paths* algorithm is similar to Dijkstra's algorithm, but stores the k shortest *paths* instead of the single previous *hop* in each node. This is necessary because in a multi-constrained environment, sub-paths of shortest paths are not necessarily shortest paths themselves [59]. If a fixed value for k is specified, the multi-constrained shortest path from a source to a destination may not be found. For SAMCRA, the number of paths stored in each node is unrestricted, meaning that *all* possible paths may need to be stored before the shortest one can be selected. This property leads to the worst-case NP-complete behavior of SAMCRA [61].

The *non-dominance* concept allows for a drastic reduction of the number of sub-paths that need to be stored in the router nodes. This optimization dismisses newly computed sub-paths from the source to the current routing node that a priori lead to a non-optimal path from the source to the final destination, based on previously computed sub-paths stored in the node. More concrete, new sub-paths between the source and the current routing node are dismissed if a previous sub-path to the same routing node has a weight vector for which each vector component is smaller than the corresponding component of the new sub-path weight vector.

The *look-ahead* extension further reduces the search space of possible paths by predicting the total length from source to destination for each sub-path stored in intermediate routers. This path length prediction is based on the sub-path history and Dijkstra shortest path information for all constraints from the intermediate router to the final destination. By applying the *look-ahead* extension further reduces the search space of possible paths by predicting the total length from source to destination for each sub-path stored in intermediate routers. This path length prediction is based on the sub-path history and Dijkstra shortest path information for all constraints from the intermediate router to the final destination. By applying the look-ahead concept, sub-paths with the lowest end-to-end predicted path length are evaluated *first*. Meta-code for the complete algorithm and the corresponding discussion can be found in Appendix F.

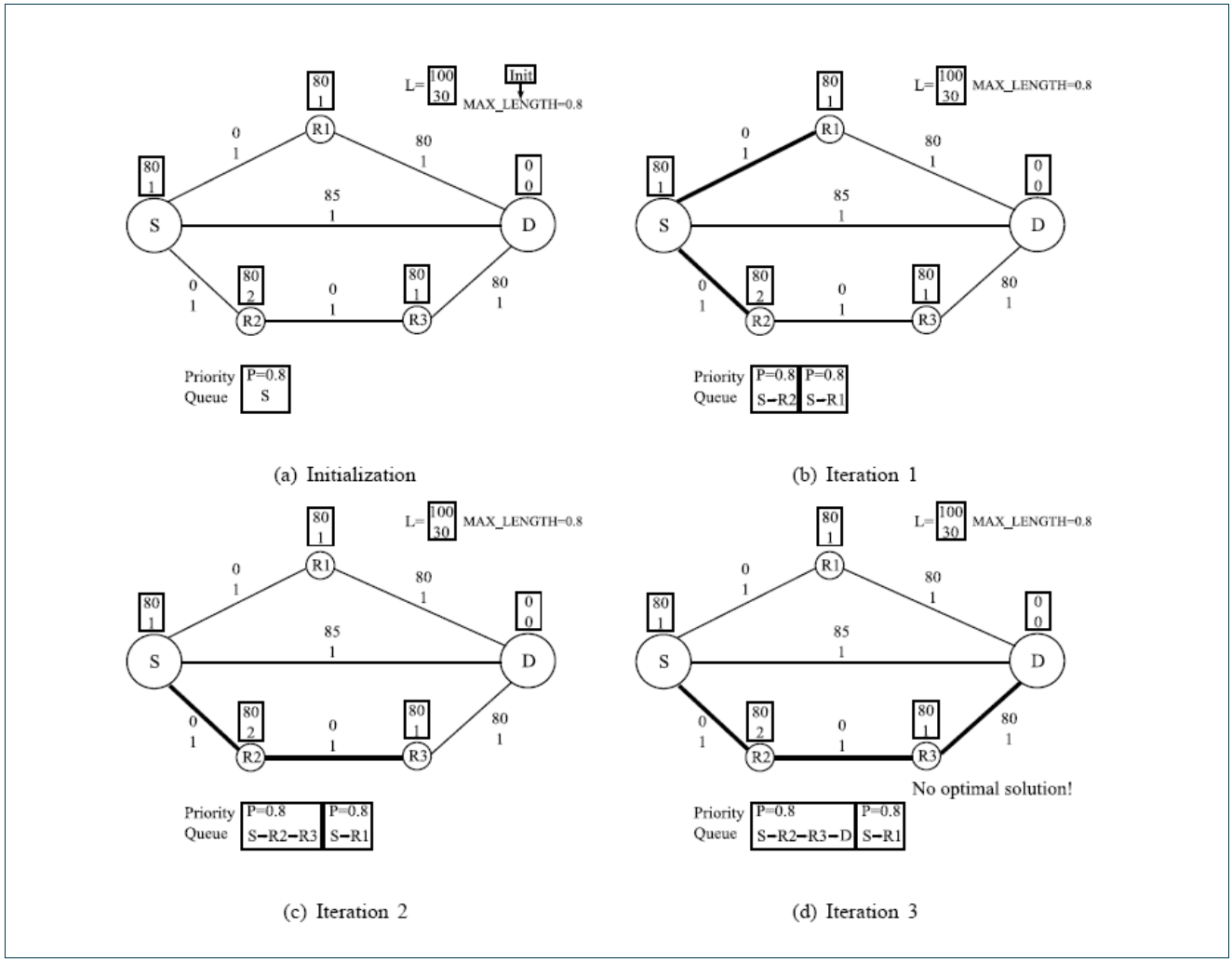


Figure 40 - Example scenario where the SAMCRA algorithm with look-ahead information computes a sub-optimal path from source *S* to destination *D* (*look-ahead* information is provided in the rectangle above each node). Items in the priority queue are listed in the order in which they will be dequeued in the next iteration. In theory, items with equal priorities have equal probabilities of being dequeued in the next iteration. Bold paths depict new sub-paths added to the priority queue in the current iteration.

The two-dimensional SAMCRA routing example depicted in Figure 40 shows the q-norm length definition (see Eq. (37) does *not* guarantee that the SAMCRA algorithm can find the optimal solution, however. The two-dimensional link weights are depicted above each link and look-ahead information for each node is provided in a rectangle above the node. The routing constraints are provided in the upper right corner of each iteration step. During the initialization phase of the path computation from the source node *S* to the final destination node *D* (Figure 40(a)), node *S* is added to the priority queue with a priority equal to the predicted path length to destination *D*. Then, during the first iteration (Figure 40(b)), both paths *S* → R1 and *S* → R2 are added to the priority queue (predicted length equals 0.8) as being plausible sub-paths of the final solution. Because *all items in the priority queue have equal priorities (lengths)*, the scenario depicted by Figure 40(c) and (d) can arise, resulting in the sub-optimal SAMCRA path *S* → R2 → R3 → *D*. The path via edge *S* → R1—which is part of the

optimal solution—is never explored because the SAMCRA algorithm halts the first time a complete path from source node S to destination node D is dequeued from the priority queue (see Appendix F.1, lines 14–15), thinking it has found the optimal solution.

```
1   for i = 1 to n do
2           a_i ← a_i/L_i
3           b_i ← b_i/L_i
4   sort components of a in descending order
5   sort components of b in descending order
6   for i = 1 to n do
7           if a_i < b_i
8           then return a has highest priority
9           if a_i > b_i
10          then return b has highest priority
11  return a and b have equal priorities
```

Figure 41 – Weight vector comparison function

Fortunately, the SAMCRA algorithm can be adapted to cope with this issue. Instead of using Eq. (37) to determine the order in which sub-paths will be evaluated (remember that SAMCRA uses predicted path length as the key for the priority queue), the information contained in the entire *path weight vector* should be used when ordering the sub-paths. Two path weight vectors $a=V(p)$ and $b=V(p')$ (where V( ) is the path vector of the path) can then be compared by the algorithm shown in Figure 41. Applying this function guarantees that optimal subpaths are evaluated *first*. Because the length information is not lost for the $n - 1$ non-dominating dimensions, this path ordering method can take them into account. This adaptation is particularly important when the SAMCRA algorithm is applied in a hop-by-hop scenario [62], as this requires an exact solution in each intermediate node in order to prevent *routing loops*.

4.1.1.3 *Performance Evaluation Results*

Simulation Model

In this section, we present performance evaluation results for the network-based server selection mechanism introduced above. As an example network application for simulation, we chose highly interactive online gaming. At present, this is one of the most-popular resource-hungry Internet applications and it is representative for the problem statement described in the introduction. From a server selection perspective, it is important to select a

server that has enough free resources to service the new user session. Additionally, interactive games are delay-sensitive, so the end-to-end delay between the client and server should be minimized. MacKenzie and Ware [63] have shown that a lag around or above 75ms degrades human performance in motor-sensory tasks on interactive systems. Therefore, we decided the total path delay should be less than or equal to 30ms, yielding a maximum round-trip time (RTT) of 60ms. This leads to a multiple constraints routing problem, characterized by both server load and network delay. The performance of a server selection mechanism is then represented by the maximum number of concurrent sessions it can support for a specified network and server capacity. Sessions can be rejected either because they are sent to a server that has reached maximum capacity or because the computed path from client to server cannot meet the delay constraint.

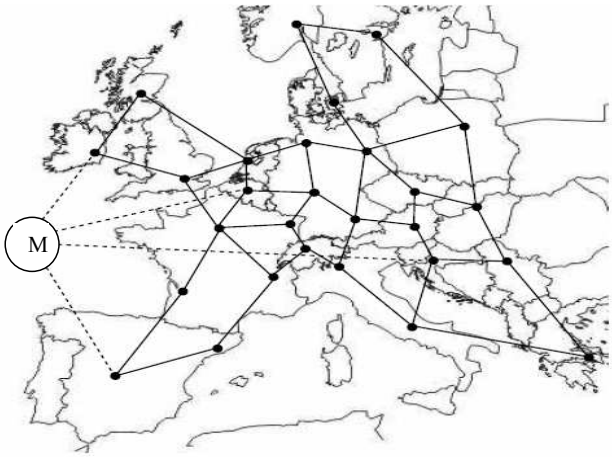


Figure 42 - Network topology used for simulation. The network consists of 28 regular nodes interconnected by 41 links. An anycast group *M* consisting of four servers is represented by a single logical node connected to four distinct routers (selected at random during simulation).

We consider a pan-European reference network topology [64], depicted in Figure 42, to perform the simulations. This topology consists of 28 regular router nodes, interconnected by 41 network links, and an anycast group of 4 servers, each attached to a different router node. The anycast servers could be considered either as regular gaming servers or as a remote rendering service for the resource-constrained devices. Each link is characterized by a link delay value (depending on its length) and a discrete number of sessions Δ it can support, assuming that bandwidth requirements are equal for all sessions. The advertised link delay for a link already carrying Δ sessions is set to ∞, thereby avoiding that new sessions are routed via the congested link. On the server side, each server is characterized by the number of sessions π that can be handled simultaneously. Consequently, the advertised load for a server handling π sessions equals π/Π. For simulation purposes, we assume all links have the same capacity Δ (expressed in sessions/link) and all servers are capable of handling $\Pi = 20$ sessions. For 4 servers, this results in a maximum of 80 concurrent sessions.

Each session is specified by its duration time. According to Feng et al. [65], the probability density function of measured session duration times for highly interactive games can be matched to a two-parameter Weibull distribution:

$$f(T) = \frac{\beta}{\eta}\left(\frac{T}{\eta}\right)^{\beta-1} e^{-\left(\frac{T}{\eta}\right)^{\beta}} \quad (38)$$

where β is the slope of the distribution and η is a scale parameter. They estimated β = 0.5 and η = 20 when session duration is expressed in minutes. During the simulation runs, session duration times were randomly drawn from this distribution. Since the expected value for a Weibull distribution is given by:

$$E(T) = \eta\,\Gamma\left(1+\frac{1}{\beta}\right) \quad (39)$$

the average session duration equals 40 minutes. Note that Γ() is the Gamma function. A last simulation parameter that needs to be specified is the average session request generation frequency λ. This parameter represents the parameter of the Poisson process generating the new session requests and together with the session duration time, it specifies the total server (and network) load. We chose λ = 1.5 (sessions/minute), which corresponds to an average total server load of 75%, if all session requests are accepted.

Evaluation Results

During the performance evaluation, four different routing algorithms are compared. These algorithms are summarized in Table 15.

Name	Description
Delay Only	Dijkstra shortest path routing based on link delay values.
Best Server	First the server with the lowest system load is selected, then the Dijkstra shortest path based on link delay values is computed.
SAMCRA	Exact source-based multiple constraints routing using server load and path delay as routing metrics.
Hop-by-hop SAMCRA	Each router computes the SAMCRA path from the router to the destination [62]. This avoids source-based routing, thus leading to a more scalable solution than regular SAMCRA (due to the static routing tables). Because subsections of shortest paths in multiple dimensions are not necessarily shortest paths themselves [62], only a sub-optimal path is computed, however.
Maximum flow	This optimal, offline technique (due to Ford and Fulkerson) determines the maximum amount of flows between a given source and destination. It essentially locates paths between source and destination with free capacity (referred to as augmenting paths), and routes as many flows as possible

	over these paths. Similar to SAMCRA, supporting the anycast scenario also requires the incorporation of a virtual resource, whereby the capacity of the virtual links is proportional to the processing rate of the attached resource. In case job characteristics of individual clients (e.g., required processing capacity and average runtime) remain identical, a virtual source can be introduced in the network, together with links connecting the virtual node to the physical clients. Virtual link capacities are proportional to the job arrival rate of the attached client, and the classical, single-commodity maximum flow algorithm can be employed. However, in case job characteristics differ between clients, a virtual client cannot be introduced and a multi-commodity, maximum flow algorithm needs to be used between all clients and the single, virtual destination. The remainder of this section only considers the single commodity, maximum flow algorithm. Finally, the incorporation of a deadline as job constraint causes the pseudo-optimal behavior of the maximum flow technique. Indeed, paths violating the deadline constraint are not considered as a possible augmenting flow path, and thus the true maximum flow is not attainable.

Table 15: Algorithms used for multi-constrained scheduling and routing

The simulations are run for a network link capacity Δ ranging between 0 and 20 sessions per link. For each capacity Δ_i, a thousand simulation runs of thousand minutes are executed, where the anycast server locations are selected at random for each simulation run. Within each simulation, every time a new session is generated, a source router (i.e., one of the 28 regular routers) is assigned at random. All algorithms were tested with the same random generator seeds, in order to guarantee an equivalent simulation environment.

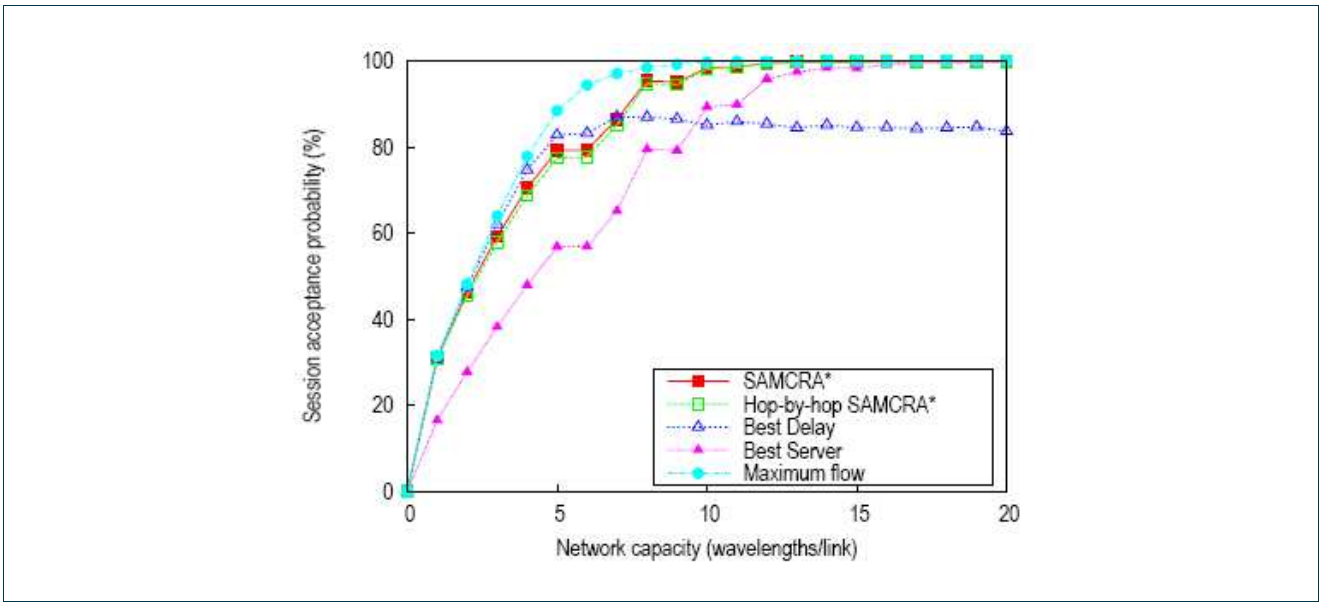


Figure 43 – Session acceptance probability vs. network capacity for a fixed average resource load of 75%

The session acceptance probability for each of the four algorithms is depicted in Figure 43. When the network capacity is limited, *delay only* routing performs best. However, as the network capacity increases, the average session acceptance rate stagnates because the server load is neglected for taking the routing decisions. The worst-case scenario—indicated by the minimum acceptance rate—illustrates that the performance of this routing strategy is highly dependent on the anycast server placement. The next algorithm, *best server* routing, converges slowly to an average acceptance probability close to 100% for an increasing network capacity. Due to the inefficient allocation of network resources, a high acceptance rate can only be achieved for an over-dimensioned network, however. *SAMCRA* routing, both source-based and applied hop-by-hop, clearly combines the advantages of both the *delay only* and *best server* approach by providing an optimal solution for each network capacity. Moreover, also in the worst-case scenario, *SAMCRA* performs significantly better than the other routing algorithms. In theory, *hop-by-hop SAMCRA* provides sub-optimal results, but in practice its performance appears to approximate that of *SAMCRA*.

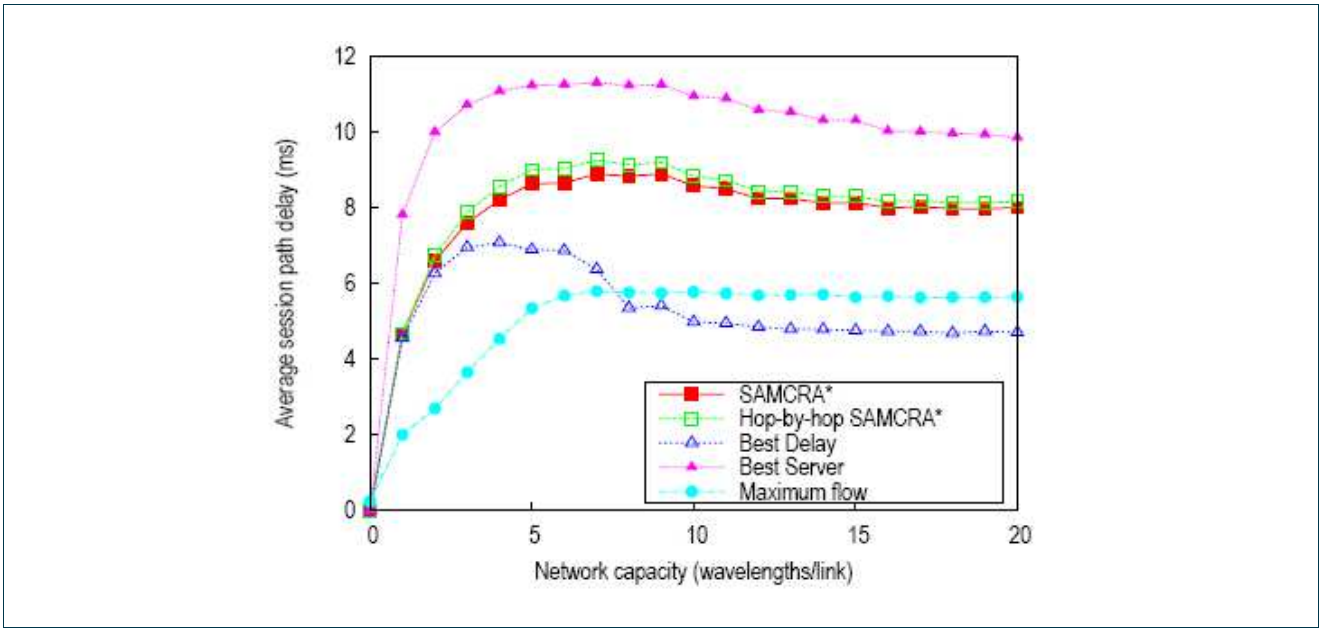


Figure 44 – Average session delay vs. network capacity for a fixed average resource load of 75%

Figure 44 shows the minimum, average and maximum path delay for accepted sessions, again as a function of the network link capacity. As expected, *delay only* routing performs best, with an average path delay that is half as long as the average delay of the *best server* approach. *SAMCRA* steers a middle course, with an average path delay between 8 and 9ms. The performance evaluation results clearly indicate the added value of multiple constraints routing for network-based server selection. Furthermore, this approach optimizes the routing decision for an arbitrary number of routing criteria, so it is not limited to the two-constraint optimization problem considered in this performance evaluation. *SAMCRA* belongs to the class of NP-complete problems, but throughout the simulations we did not experience a single routing instance leading towards intractability.

4.1.2 Joint Communication and Computation Task Scheduling in Grids: a Multicost Approach

In order for Grid systems to be used in real world commercial applications and demanding scientific experiments, end-to-end Quality of Service (QoS) is desirable. The main approach to providing end-to-end QoS is reservations and especially reservations that are performed "in-advance". Reservations in Grids expand in various domains, such as computation, communication, storage resources and even instruments. Thus, the efficiency of a Grid system depends on the development of sophisticated resource management systems capable of allocating resources based on user QoS requirements.

We propose an algorithm that jointly addresses the communication and computation problems. In comparison to the solutions proposed in [46],[47] our algorithm also uses advance reservations in the scheduling of the communication resources (similar to [48]). 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 (computation resource) in the form of a session or 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 (session or burst) 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.

The proposed algorithm consists of three phases: (a) 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. (b) Secondly, the algorithm obtains the set of candidate non-dominated (path, cluster) pairs from the source to all clusters that can process the task. (c) Finally, the algorithm chooses from the previously computed set a pair that minimizes the task completion time, or some other performance criterion. The proposed algorithm is designed for a distributed architecture, but can be easily extended to function in a centralized approach. 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.

In Section 4.1.2.1 we present the utilization profiles of the communication and computation resources. In Section 4.1.2.2 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 4.1.2.1. The joint communication and computation scheduling algorithm is presented in Section 4.1.2.3 and its extensions in Section 4.1.2.4. Section 4.1.2.5 presents performance results.

4.1.2.1 *Communication and Computation Utilization Profiles*

Link Utilization Profiles

For the sake of being specific, we assume that the network connecting the clusters, the schedulers and the data repository sites follows the Optical Burst Switching (OBS) paradigm [54],[56]. This does not limit the applicability of our algorithms, which can be used in any network supporting advance reservations as discussed in [50].

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 [57]. In the OBS paradigm, each node needs to keep a record of the capacity reserved on its outgoing links as a function of time [58] in order to perform channel scheduling and reservations. Assuming each session or burst reserves bandwidth equal to the link capacity for a given time duration (the case of WDM networks does not fall in this category, but can be treated similarly as discussed in [50]), the *utilization profile* $U_l(t)$ of 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 session or burst. We define the *capacity availability profile* of link l of capacity C_l as $C_l(t)=C_l-U_l(t)$. In order to obtain a data structure that is easier to handle in an algorithm, we discretize $C_l(t)$ in time steps of duration τ_l to obtain the *binary capacity availability vector* $\hat{C}_l$, abbreviated CAV, as the vector whose k-th entry is:

$$\left\{\hat{C}_l\right\}_k = \left\{\begin{matrix} 1, \text{ if } C_l(t)=1 \\ 0, \text{ otherwise} \end{matrix}\right\}, \text{ for all } (k\text{-}1)\cdot\tau_l \le t \le k\cdot\tau_l, k=1,\ldots,d^l \qquad (40)$$

where d^l the dimension of the CAV (see Figure 45).

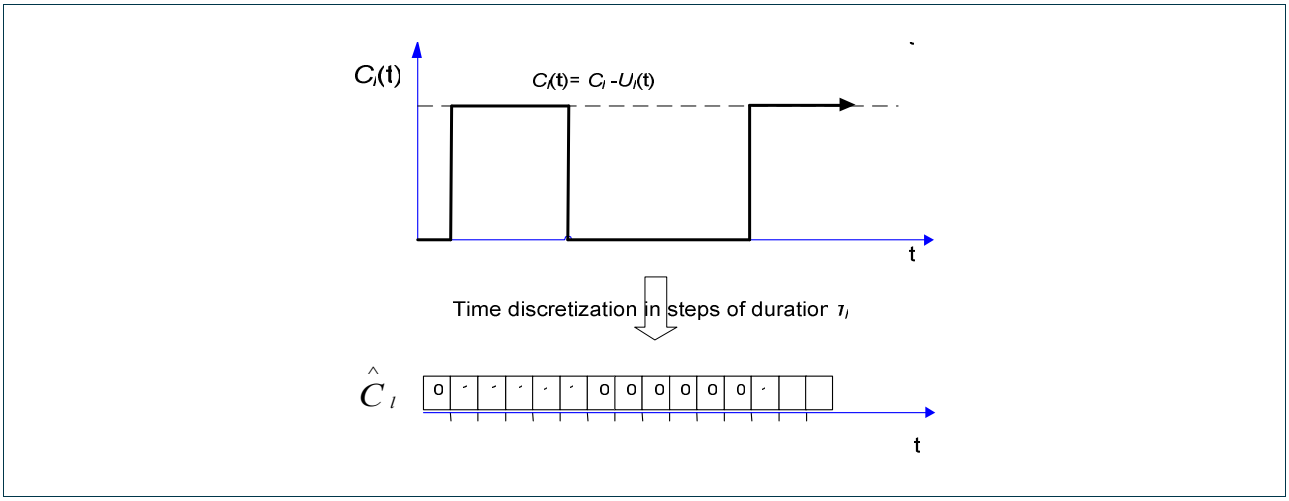


Figure 45 – The capacity availability profile $C_l(t)$, and the binary capacity availability vector $\hat{C}_l$ of a link l of capacity C_l.

Clusters Utilization Profiles

We assume that a computation resource *m* is a cluster that consists of W_m CPUs of equal processor speed H_m (measured, e.g., in MIPS), so that the total computation capacity of cluster *m* is $C_m=W_m \cdot H_m$. We also assume that when a task starts executing at a CPU it cannot be preempted. A task requests to be executed in *w* CPUs ($w <= W_m$), 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 *w* (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 w-cluster availability vector* $\hat{W}_m(w)$**,** as follows:

$$\left\{\hat{W}_m(w)\right\}_k = \left\{\begin{matrix} 1, \text{if } W_m - U_m(t) > w \\ 0, \quad \text{othewise} \end{matrix}\right\}, \text{for all } (k-1)\cdot\tau_m \le t \le k\cdot\tau_m, k = 1,2,...,d^m \quad (41)$$

where d^m is the maximum size of the $\hat{W}_m(w)$ vector (see Figure 2).To simplify presentation, we assume for this study that each task requests *w*=1 CPUs, which is the most usual case. Then, we can denote $\hat{W}_m(w)$ 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 the communication and computation resources and to separately control the efficiency-accuracy we want to obtain in each case.

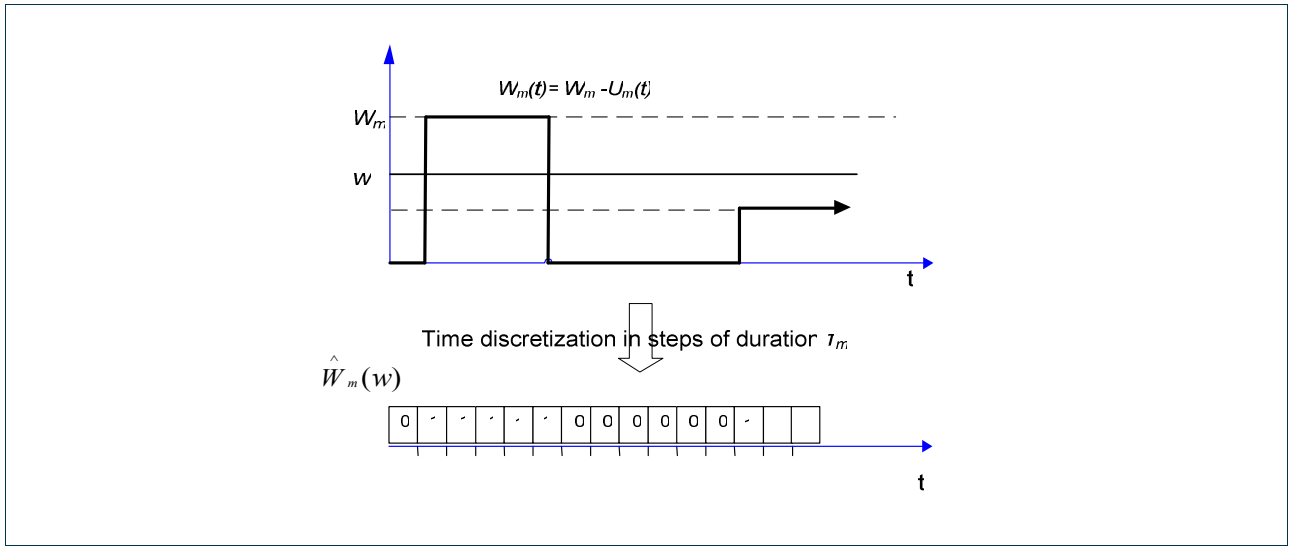


Figure 46 – The cluster availability profile $W_m(t)$, and the binary *w*-cluster availability vector $\hat{W}_m(w)$ of a cluster m.

Utilization Profiles in a Distributed Architecture

In a distributed architecture, each distributed scheduler maintains a "picture" of the utilization of all network and computation resources, which can be different among the distributed schedulers, because of non-zero network propagation delays. In our approach, this is done by maintaining a utilization database with capacity and cluster availability vectors for all the links and clusters. Update information (in the form of update messages) is communicated among nodes to synchronize the locally maintained profiles with the actual utilization.

4.1.2.2 *Task Routing and Scheduling Problem under Consideration*

We are given a Grid infrastructure consisting of a network with links $l \in L$ of known propagation delays d_l, and capacity C_l and a set M of computation resources (clusters). Cluster $m \in M$ has W_m CPUs of a given processor speed H_m (eg, in MIPS). A task is created by a user with specific needs: input data size I (bits) and computational workload h (MIs). The user communicates this information to its attached distributed scheduler S [49]. 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 47 presents an instance of the problem.

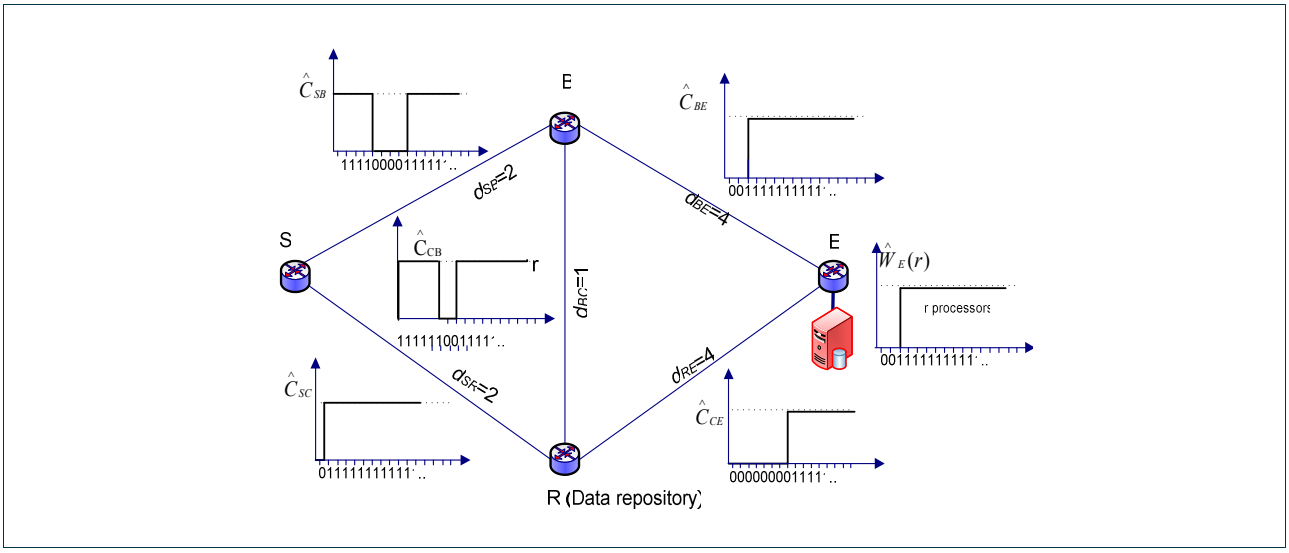


Figure 47 – A task request is forwarded to the distributed scheduler S. The task requires the transmission of a burst with duration $b_l=I/C_l$ from source (which can be S or a data repository site) and w CPUs to execute. Each link is characterized by its propagation delay (in τ_l time units) and its binary capacity availability vector. Node $E \in M$ has a cluster with binary w-cluster availability vector $\hat{W}_E(w)$.

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 section 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 (next paragraph) 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 [50]. For example, for the topology of Figure 47, the CAV of path *SBE* (p_{SBE}), consisting of links *SB* and *BE*, is

$$\hat{C}_{SBE} = \hat{C}_{SB} \oplus \hat{C}_{BE} = \hat{C}_{SB} \,\&\, \mathrm{LSH}_{2 \cdot d_{SB}}(\hat{C}_{BE}) \quad (42)$$

where $\hat{C}_{SB}$ and $\hat{C}_{BE}$ are the CAVs of links *SB* and *BE,* respectively, and $\mathrm{LSH}_{2 \cdot d_{SB}}$ defines the left shift of $\hat{C}_{BE}$ by $2 \cdot d_{SB}$ (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 48.

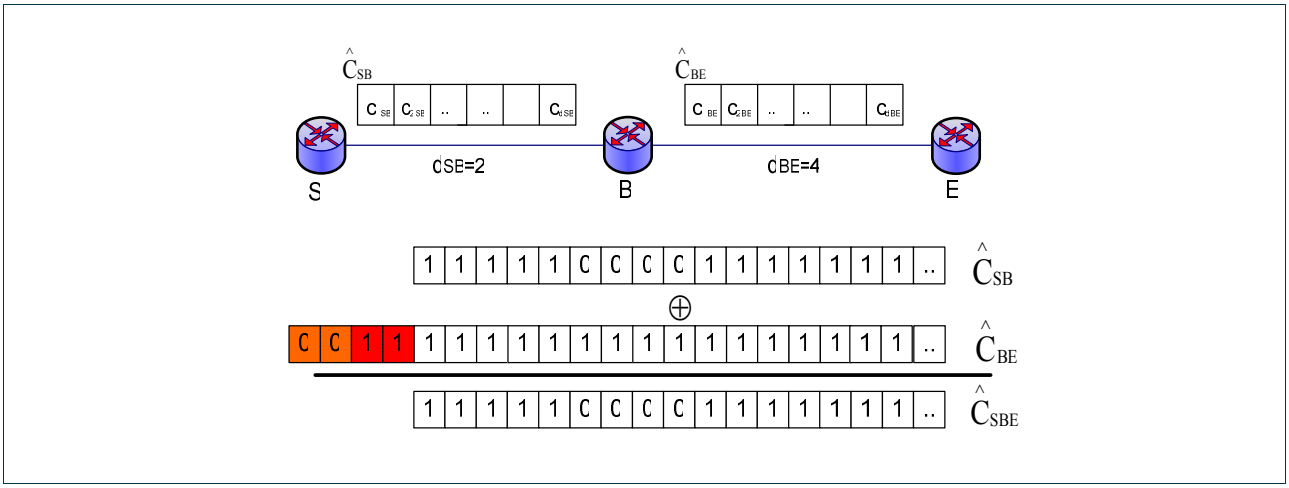


Figure 48 – Calculation of the path capacity availability vector $\hat{C}_{SBE}$. $\hat{C}_{BE}$ is shifted by $2 \cdot d_{SB}$ τ_l-time units (d_{SB}=2 in this example), before the AND operation is applied.

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_l ($b_l = I / C_l$ where I is the input data size) over the path *p* in order to execute it at cluster *m*. We define $R_p(b_l)$ as the first position after

which $\hat{C}_p$ has b_l consecutive ones. In other words, $R_p(b_l)$ is the earliest time after which a burst of duration b_l can start its transmission on path p. The earliest time that the task can reach cluster m is given by $\mathrm{EST}(p,b_l)=R_p(b_l)+b_l+d_p$. 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 $\hat{W}_m(p,b_l)=\mathrm{MUV}_{\mathrm{EST}(p,b_l)}(\hat{W}_m)$ gives the time periods that S can schedule the task over path p at cluster m.

With respect to Figure 47 and Figure 48, we assume that we want to transmit a task of duration b_l=3 from S to the cluster at node $E \in M$, over path p_{SBE} with propagation delay d_{SBE}=6. The capacity availability vector $\hat{C}_{SBE}$ was calculated in 0, and we have $R_{p_{SBE}}(b_l)=0$. The task reaches E after $\mathrm{EST}(p_{SBE},b_l)=R_{p_{SBE}}(b_l)+b_l+d_{SBE}=9$. Also, S has a (possibly outdated) knowledge of the cluster availability profile $\hat{W}_E$. The cluster availability vector that gives the time periods that S can schedule the task at E is $\hat{W}_E(p_{SBE},b_l)=\mathrm{MUV}_{\mathrm{EST}(p_{SBE},b_l)}(\hat{W}_E)=\mathrm{MUV}_9(\hat{W}_E)$, which is the operation of setting the 9 first entries of vector $\hat{W}_E$ to zero. This process is depicted in Figure 49.

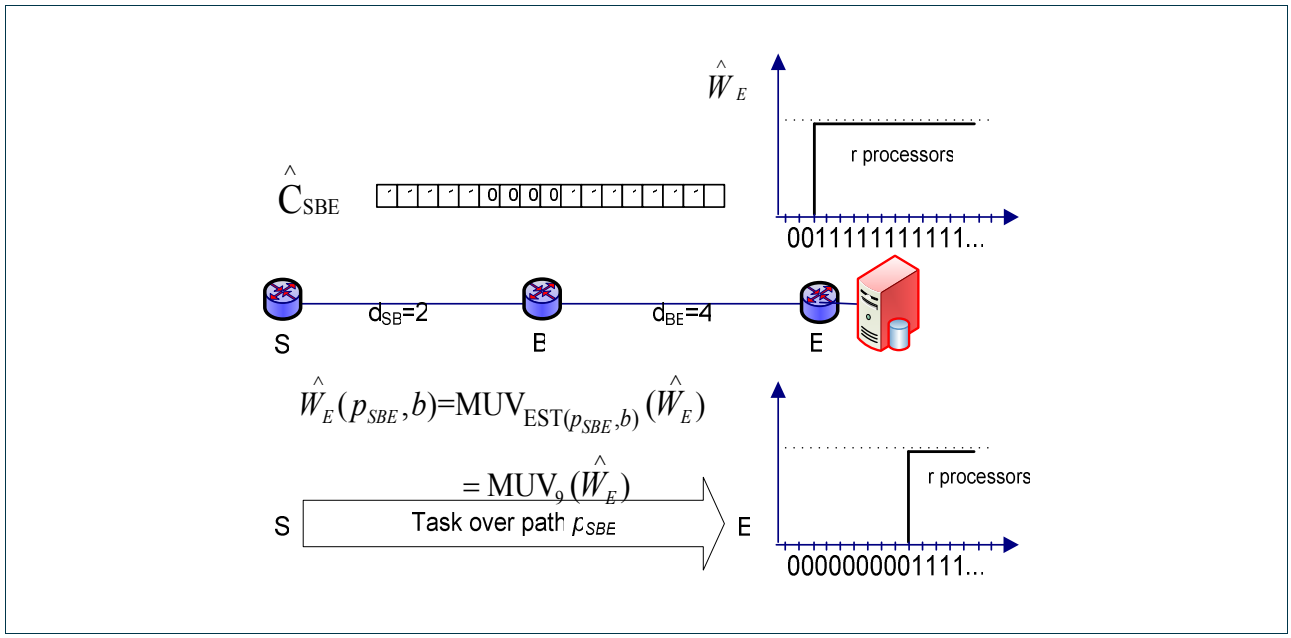


Figure 49 – Scheduler S wants to transfer a task of data duration b_l=3 over path p_{SBE}. We denote by $EST(p_{SBE},b_l)$ the earliest time the task can reach E over p_{SBE} and by $\hat{W}_E(p_{SBE},b_l)$ the cluster availability vector that gives the time periods at which S can schedule the task at E. To calculate $\hat{W}_E(p_{SBE},b_l)$, we put 0's in the first $EST(p_{SBE},b_l)$=9 elements of $\hat{W}_E$.

4.1.2.3 *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: (a) given that the input data are located at source (which can be the scheduler S or a data repository site R) we first calculate the set $P_{n\text{-}d}$ of non-dominated paths between the source and all the network nodes. (b) 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. (c) 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.

(a) 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 of interest (such as number of hops, the number of executed tasks in a cluster, etc). A cost vector [51] can then be defined for a path p consisting of links 1,2,…,k, based on the cost vectors of its links, according to

$$V(p) = \mathop{\mathrm{e}}_{l=1}^{k} V_l \overset{def}{=} \left(\sum_{l=1}^{k} d_l, \mathop{\oplus}_{l=1}^{k} \hat{C}_l \right), \tag{43}$$

where $\oplus$ is the associative operator defined in Eq. (42)

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 dominates p_2 (notation: $p_1 > p_2$) **iff**

$$\sum_{l \in p_1} d_l < \sum_{l \in p_2} d_l \text{ and } \mathop{\oplus}_{l \in p_2} \hat{C}_l \le \mathop{\oplus}_{l \in p_1} \hat{C}_l, \tag{44}$$

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 now formally described, for the case where the link cost vectors consist of one additive parameter, and the binary capacity availability vector of the link.

We denote by V_l the cost vector of link l. Each path is represented by a label that includes the cost vector associated with it and the first hop to the source using that path. The source that serves the connection is taken to be node S.

We let O_i be the set of labels of the paths from node S to a node n_i, and $O = \cup_{i \neq S} O_i$ be the set of all labels. Initially, every node has a single label corresponding to the link (if any) that connects it directly to the origin node. In each of the following steps, the algorithm marks labels (equivalently paths) from the set O as final. We let $O^d \subseteq O$ be the subset of all final labels for all the nodes, and $O_i^d \subseteq O_i$ be the set of final labels for node n_i. We also let T be the set of nodes with at least one final label. The algorithm can now be described as follows:

Step 0 (Initialization): $O=\{Vp_1, Vp_2, \ldots, Vp_N\}$, $O^d=\{\}$, $T=\{\}$, where Vp_i is the label of the path p_i (if any) leading directly from node S to node n_i, and $n_i \neq S$.

Step 1 (Choosing the optimum label): The label of path p whose cost vector minimizes the additive component is chosen. In case of a tie, we look at the second component, which is the binary capacity availability vector, and a dominant one is chosen. If Vp_i is the cost vector of the chosen label and n_i is the corresponding node to which it leads, then the following updates are performed:

$$O_i^d = O_i^d \cup \{V_{p_i}\},\ O^d = O^d \cup \{V_{p_i}\},\ T = T \cup \{i\}.$$

Step 2 (Obtaining the new labels): The neighbors of node n_i, which may or may not belong to the set T, are now considered and are given new labels (except for the origin node and the node specified as the previous node in the label). The new label for the path p_j leading to the neighbor n_j of node n_i by extending the path p_i through the link $l=(n_i,n_j)$ is then computed in the following way. The new cost vector is updated according to $V_{p_j} = V_{p_i} \oplus V_l$ where V_l is the label of the link $l=(n_i,n_j)$ and "$\oplus$" represents the operation defined in Eq. (43).

Step 3 (Discarding dominated paths): Every neighbor considered in step 2 compares its new label with its previous labels using the domination relation defined in Eq. (44). Let n_j be one of the neighbors of node n_i, Vp_j the new label obtained from step 2 and O_j be the set of labels for this node. The new label has to be compared with the labels in O_j (both final and non-final). If any cost vector in O_j dominates Vp_j, then Vp_j is discarded and O_j does not change. If the new cost vector Vp_j is not dominated by any of the vectors in O_j , then Vp_j is added to the set O_j and O, so that $O_j = O_j \cup \{V_{p_j}\}$ and $O = O \cup \{V_{p_j}\}$. If the new vector dominates one of the vectors in O_j, then O_j and O are updated by eliminating the dominated vectors and adding the new vector Vp_j. Note that it is not possible for the new vector to dominate an existing vector and be dominated by another one at the same time.

Step 4 (Termination): If after an iteration the set O^d is equal to O, the algorithm is completed. Otherwise, (when there are still some labels to be chosen) we go back to Step 1.

The set $P_{n\text{-}d}$ of non-dominated paths from the given source S to all destinations is the final set O^d.

(b) 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:

$$V(pm) = \left(V(p),\ \hat{W}_m(p,b_l)\right) = \left(\sum_{l=1}^{k} d_l,\ \bigoplus_{l \in p} \hat{C}_l,\ \hat{W}_m(p,b_l)\right), \quad (45)$$

where $\hat{W}_m(p,b_l)=\mathrm{MUV}_{\mathrm{EST}(p,b_l)}(\hat{W}_m)$ is the binary cluster availability vector of m with 0's at the first EST(p,b_l) elements (Section 4.1.2.2).

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 dominates p_2m_2 (notation: $p_1m_1 > p_2m_2$) iff

$$p_1 > p_2 \text{ and } \hat{W}_{m_1}(p_1, b_l) \leq \hat{W}_{m_2}(p_2, b_l) \ , \qquad (46)$$

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. (46) to the elements of $P_{n\text{-}d}$.

(c) 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 1: 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 $R_i(b_{m,i})$ after which $\hat{W}_{mi}(p_i, b_l)$ has $b_{m,i} = h/H_{mi}$ consecutive ones. In other words, $R_i(b_{m,i})$ is the earliest time at which a task of computation workload h can start execution on m_i. Note that the way $b_{m,i}$ is calculated accounts for the computation capacity of resource m_i, and that $\hat{W}_{mi}(p_i, b_l)$, by definition, accounts for the earliest transmission time, the propagation delay of path p_i and the transmission delay (Section 4.1.2.2).

Step 2: Select the cluster with the minimum task completion time

Select the pair p_im_i that results in the minimum completion time $R_i(b_{m,i}) + b_{m,i}$ 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 $R_i(b_{m,i})$.

Step 3: Selecting the time to schedule the burst

Having chosen the pair $p_i m_j$ we transmit the task at the earliest time possible. The time offset TO_{path} for the task transmission is $R_{p_i}(b_l)$, defined as the first position after which $\hat{C}_{p_i}$ has b_l consecutive ones (like in Section 4.1.2.2).

Step 4: 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 [50] and [52], 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 $\hat{W}_{mi}(p_i, b_l)$, $R_l(b_{m,i})$, and $R_{p_i}(b_l)$ to take into account the round trip time before the data transmission.

4.1.2.4 *Improvements of the Multicost Algorithm*

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

A serious drawback of the algorithm described previously 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. (44).

In [50] 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* (AW) of the link, as weight($\hat{C}_l$), where the weight( ) of a binary vector represents the total number of 1's in the vector.

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 4.1.2.3.a. The algorithm still maintains the binary vectors of the paths but the domination relationship that is used to prune the paths is not Eq. (44) but the following:

p_1 pseudo-dominates p_2 ($p_1 >_{ps} p_2$) iff

$$\sum_{l \in p_1} d_l < \sum_{l \in p_2} d_l \quad \text{and} \quad weight(\bigoplus_{l \in p_1} \hat{C}_l) > weight(\bigoplus_{l \in p_2} \hat{C}_l) \tag{47}$$

When the domination relationship of Eq. (47) is used, an upper limit on the number of non-pseudo-dominated paths per source-destination pair is the dimension d' 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'+1-dimensioned cost vector into a cost vector of dimension 2 that conveys most of the important information contained in the original vector.

Joint Communication and Computation Task Scheduling Algorithm when the Task Computation Workload is not Known in Advance

In the previous sections we have assumed that the task computation workload is known in advance, and used this information in order to obtain the optimum *pm* pair for the task. We modify the proposed algorithm in order to make it applicable in the case that the computation workload of a task is not known in advance.

When the computation workload of a task is not known, we don't have to use the cluster availability vectors that depend on the time (Section 4.1.2.1). Instead, the cluster availability can be mapped into a scalar parameter. The scalar, which we will note with S_m, is the available number of CPUs on cluster *m*. S_m is given by: W_m minus the number of tasks executed and queued at *m*. Thus S_m can take negative values (in the case that all CPUs are full and more tasks are queued for execution).

The cost vector V(*pm*) (Eq. (45)), is now given by:

$$V(pm)=\left(V(p_1),\ S_m\right) \tag{48}$$

and the domination relationship of Eq. (46) is given by:

p_1m_1 dominates p_2m_2 (notation: $p_1m_1 > p_2m_2$) **iff**

$$p_1 > p_2 \text{ and } S_{m_1} > S_{m_2} \tag{49}$$

With respect to the algorithm: we compute the set $P_{\text{n-d}}$ of non-dominated paths with the algorithm described in Section 4.1.2.3.a. In order to obtain set of non-dominated {path-cluster} pairs (Section 4.1.2.3.b), we apply Eq. (49) to $P_{\text{n-d}}$ set (instead of Eq. (46)). Finally, we select the cluster with the largest S_m (step 1 and step 2 of Section 4.1.2.3.c) and transmit the task after the Time Offset (TO_{path}) that is calculated by step 3 of Section 4.1.2.3.c. Of course, we don't have to calculate $TO_{cluster}$.

4.1.2.5 *Performance Evaluation 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 variations, we conducted simulation experiments, assuming an Optical Burst Switched network. We have extended the ns-2 platform [53] and tested the following routing algorithms:

- Optimal multicost algorithm for the joint communication and computation task scheduling (MC-T), as presented in Section 4.1.2.3.
- AW heuristic multicost algorithm for the joint communication and computation task scheduling (AWMC-T), as presented in Section 4.1.2.4.
- Optimal multicost burst routing and scheduling algorithm (MC-B), as presented in [50]. 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 [56] 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 H_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 h Millions Instructions (so the average task execution time is $b_m=h/H_m$). Finally, the size of the input data burst was also exponentially distributed with average I Bytes (so the average burst duration is $b_l= I /C_l$). The source of the input data I was always the scheduler.

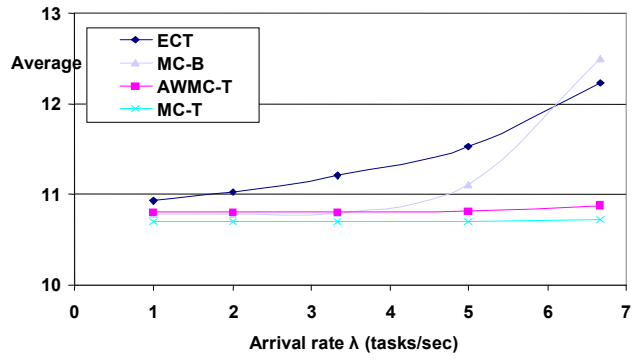
To assess the performance of the algorithms we used the following metrics:

- 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 4.1.2.3.c), due to another task that has already reserved that cluster (the so called *race* problem).

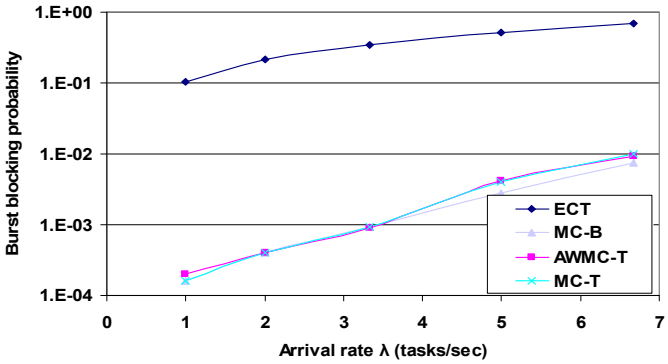
We used the following parameters: b_l =1 sec and b_m=10 sec. We classify the tasks as CPU- and data-intensive since w is considerable with respect to the total computation power, while b is considerable with respect to the total communication capacity of the network. The average total delay can take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

In Figure 50-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 50-b) and the low conflict probability (Figure 50-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 50-b) which results in increased average total delay. Similarly, the performance of the MC-B algorithm deteriorates as λ increases. The MC-B algorithm 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 50-c), which introduces additional delay to the total time of the task. From these results it is clear that in a Grid network

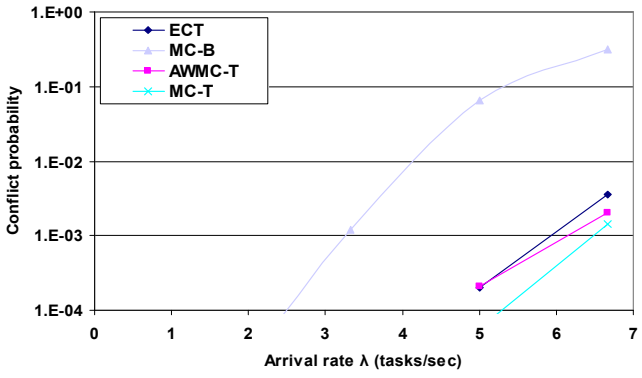
where tasks are both CPU- and data-intensive (or where some tasks are CPU-intensive and some data-intensive) performance improves significantly by jointly optimizing the use of the communication and computation resources.



(a)



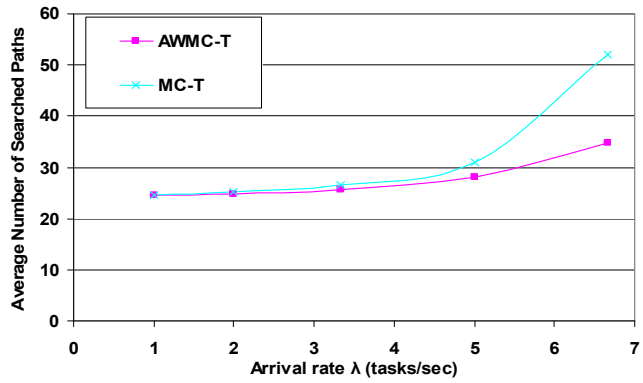
(b)



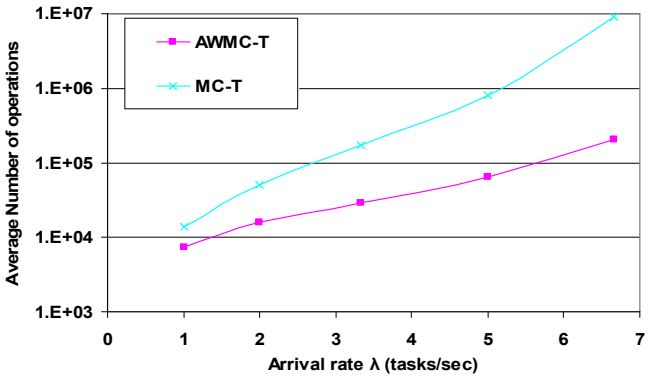
(c)

Figure 50 - Algorithms performance for tasks that are CPU- and data-intensive: (a) average total delay, (b) burst blocking probability and (c) conflict probability.

It is worth noting 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 51-a we show the average number of searched paths per task request (that is, the average size of the set $P_{n\text{-}d}$, presented in Section 4.1.2.3.a). 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 51-b shows the average number of operations required to route and schedule a task by the two algorithms (defined as the number of bitwise comparisons, additions, and AND operations required for computing the binary utilization profiles of the cost components). 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.



(a)



(b)

Figure 51 - Algorithms complexity: (a) average number of searched paths, and (b) average number of operations.

4.2 Admission and Scheduling Algorithm to Optimize the Cost of Computation, Communication and Storage Resources

A scheduling algorithm must answer the following fundamental question: Which resource set should be used when for each task? We will present an integer linear programming (ILP) model [66], which incorporates both admission control and resource allocation functions. We then investigate the behavior of our algorithms under different Grid and task loads, and pay special attention to the effects of network constraints in Grids.

Existing Grid scheduling algorithms show limited interest in bandwidth usage. For instance, James et al. present and evaluate several heuristics in [67], as do Hamscher et al. in [68], but none take network constraints into account. Recently, Ranganathan and Foster combined resource assignation and data replication algorithms in [69]. The approach presented here is different: we don't allow data replication, but instead consider the network topology a first-class entity of our scheduler.

The rest of Section 4.2 is organized as follows. We introduce our Grid model, along with various notations, in Section 4.2.1. Then in Section 4.2.2 we present our scheduling algorithms, and proceed by describing our evaluation setup. Finally; the evaluation results, analysis and conclusions are given in Section 4.2.3.

4.2.1 Grid Model

4.2.1.1 *Resources*

A Grid is composed of a set of computational resources M and a set of storage resources (data repository resource) R. Each element $m \in M$ ($r \in R$) has a processing (storage) capacity of C_m (D_r) units, and the cost for using the resource during one unit of time is given by Q_m (Q_r). Additionally, we have a set of routers B that perform the familiar store-and-forward function of a packet router. We will assume throughout this work that the forwarding capacity of each router is sufficient for all traffic sent through it. Note that the computational and storage resources also have this router functionality. Since all these resources are connected over a network, we introduce a set of link (edge) resources L. Likewise, a link resources $l \in L$ offers a bandwidth capacity of C_l units, and has a usage cost of Q_l. If we let $V = (M, R, B)$ be the set of nodes, and L be the set of links (edges), then it is straightforward to model a Grid infrastructure as the directed graph $G = (V, L)$.

4.2.1.2 *Jobs*

Let J be the set of jobs. Each job $j \in J$ is characterized by its processing rate c_j, the size of the input (output) datasets I^r_j (I^w_j), and the input (output) bandwidth b^r_j (b^w_j). Also, each job has a budget q_j, which will be paid when the job is accepted for execution. If we suppose that the input (output) bandwidth suffices to complete the input (output) data transfer, we are able to deduce the running time of the job as:

$$t_j = \max\left(\frac{I_j^r}{b_j^r}, \frac{I_j^w}{b_j^w}\right) \quad (50)$$

4.2.2 Scheduling Algorithms

4.2.2.1 *Operation*

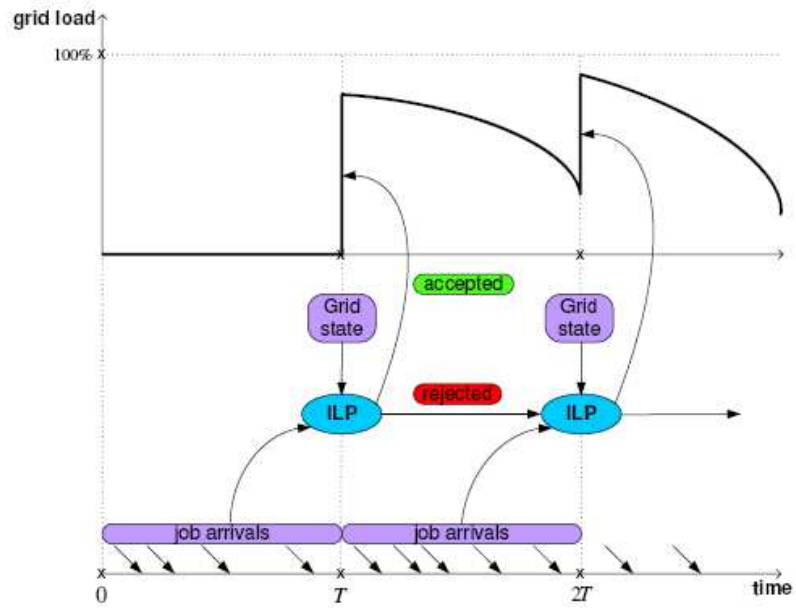


Figure 52 – Overview of the operation of the scheduler

Figure 52 details the operation of our scheduler. Scheduling is performed at periodic instants (period length *T*) in time, which allows us to react to the Grid's changing state. The ILP model incorporates two functions: job admission control and resource assignation. Thus, a solution of the ILP model tells us whether or not a job is accepted for execution, and if so, which resources it may use. Accepted jobs are all started immediately and at the same time; rejected jobs are transferred automatically to the next scheduling round. Since no jobs can be started in between scheduling rounds, a job will occupy its assigned resources during a whole number of period lengths. In effect, resources are reserved for the duration of k_j = ceil (t_j / *T*) periods, and not for the real running time of a job. Note that ceil(*x*) is the ceiling function, and returns the smallest integer that is not less than *x*.

4.2.2.2 *Variables*

In each scheduling round, we introduce three sets of *binary* variables, as summarized by .

Name	Description Takes on value of 1 iff
x_j	job *j* is executed
y_{jm}	job *j* uses computational resource *m*
y^r_{jr} y^w_{jr}	input of job *j* stored on storage resource *r* output of job *j* stored on storage resource *r*
z^r_{jl} z^w_{jl}	input of job *j* uses link (edge) *l* output of job *j* uses link (edge) *l*

Table 16 – Summary of variables for ILP scheduler

4.2.2.3 *Objective Function*

The following function expresses the cost associated with the execution of a job:

$$Q_j = \alpha \cdot k_j \sum_{m \in M} Q_m \cdot c_j \cdot y_{jm} + \beta \cdot k_j \sum_{r \in R} Q_r \left(I^r_j \cdot y^r_{jr} + I^w_j \cdot y^w_{jr} \right) + \gamma \cdot k_j \sum_{l \in L} Q_l \left(b^r_j \cdot z^r_{jl} + b^w_j \cdot z^w_{jl} \right) \quad (51)$$

The parameters α, β and γ ($0 \le \alpha, \beta, \gamma \le 1$) allow us to assign priorities to specific resource types. However, we kept all parameters constant and equal to 1 in our simulations. We now define the objective function:

$$\sum_{j \in J} \left(q_j \cdot x_j - Q_j \right) \quad (52)$$

The proposed objective function expresses the profit a scheduler will make; clearly, our scheduler is interested in maximizing this function.

4.2.2.4 *Constraints*

The first set of constraints express that an accepted job will use exactly one computation resource, the second limits the amount of usable capacity, while the third limits the amount storage capacity used:

$$\forall j \in J : \sum_{m \in M} y_{jm} = x_j \quad (53)$$

$$\forall m \in M : \sum_{j \in J} c_j \cdot y_{jm} \le C_m \quad (54)$$

$$\forall r \in R : \sum_{j \in J} (I^r_j \cdot y^r_{jr} + I^w_j \cdot y^w_{jr}) \le D_r \quad (55)$$

Edge resources (links) require an analog to the previous constraint, and a constraint which states that an accepted job uses at least one edge resource. Additionally, the association between edges and nodes is made in (analogous equations for z^{w}_{je}):

$$\forall j \in J, \forall l \in L : z^{r}_{jl} \leq x_{j} \tag{56}$$

$$\forall j \in J, \forall u \in V : \sum_{\substack{v \in V \\ (u,v) \in L}} z^{r}_{j(u,v)} - \sum_{\substack{v \in V \\ (v,u) \in L}} z^{r}_{j(v,u)} = \begin{cases} -y_{ju}, if\ u \in M \\ +y^{r}_{ju}, if\ u \in R \\ 0, otherwise \end{cases} \tag{57}$$

4.2.3 Evaluation Results

We used the Waxman model to generate 10 random network topologies. A network consists of 15 nodes, each node having a connectivity of 2. We then randomly selected 5 computational, 5 storage and 5 router resources. All resource characteristics are held constant; Table 17 summarizes their capacities and costs.

Resource type	Capacity	Cost per period
Computational	500 MIPS	1
Storage	6 Gb	1
Network	155 Mbps	1

Table 17 – Resource capacities and costs

All submitted jobs have identical characteristics; these are detailed in Table 18. It follows that the execution time of a job is t_j = 300/20 = 15 time units (see Section 4.2.2.1). Also note that the assigned budget is sufficient for execution, and that each computational resource can run at most 10 jobs concurrently. The same number of jobs can also be supported by one storage resource if both input and output of all jobs is assigned to the resource.

Parameter	Input	Output
Processing power (MIPS)	50	
Storage space (Mb)	300	300
Bandwidth (Mbps)	20	20
Budget	1000	

Table 18 – Job requirements

Job requests are generated at computational resources, and the arrivals follow a Poisson process. As it is undesirable to assign the same average number of arrivals to all arrival sites, we proceeded as follows. First, we established the maximum average number of arrivals $(\lambda T)_{max}$ for all sites. Then, we assigned each site its average number of arrivals by selecting uniformly from the interval [2, $(\lambda T)_{max}$]. Finally, each site generated job requests at their assigned average during each period

We performed evaluations for 10 random topologies, executing 5 runs for each topology. Unless otherwise stated, all evaluations have a scheduling period of T = 20 time units, which means that all jobs can run to completion within the period they were started. Simulation time is 10 scheduling periods in all cases. Jobs which are not accepted at the end of these 10 periods, are not taken into consideration for the calculation of the average queuing time. Finally, the well-known dual simplex method was used to obtain solutions for our ILP-formulated problem.

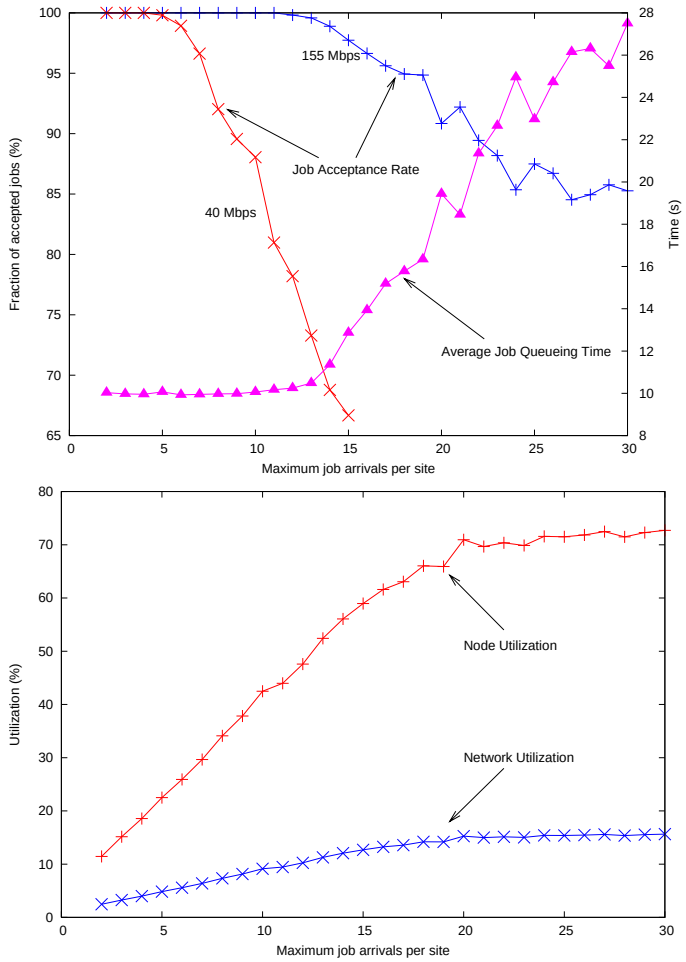


Figure 53 – Influence of $(\lambda T)_{max}$

Figure 53 shows the acceptance rate for different values of $(\lambda T)_{max}$ (as defined in Section 4.2.2.1). Satisfying all job requests is possible for much higher values of $(\lambda T)_{max}$ in the case of C_l = 155 than for C_l = 40 Mbps. The average queuing time (taken for C_l = 155) shows an increasing trend for larger values of $(\lambda T)_{max}$. For a low number of arrivals, we get an average queuing time of 10 time units (50% of the period length), which means that all jobs get executed at the first possible scheduling round. Figure 53 also shows the utilization of both node and edge resources; the node utilization for C_l = 155 Mbps approaches 75%, which is exactly the ratio of the job execution time and the period length.

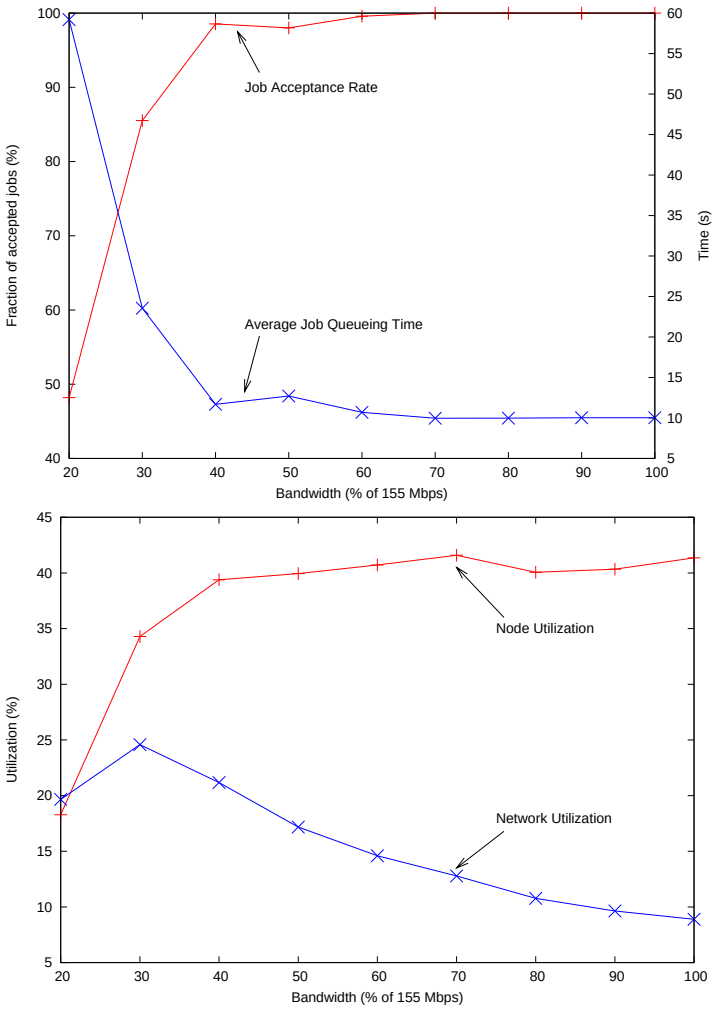


Figure 54 – Influence of available bandwidth

We start by establishing $(\lambda T)_{max}$ = 10. Figure 54 shows both the acceptance rate and the average job queuing time. We see here that a minimum of 40% of 155 Mbps is necessary to support the given load; going below this value drastically reduces performance. Also, the node utilization is constant and a little over 40% (see also Figure 53 for $(\lambda T)_{max}$ = 10) when at least 40% of 155 Mbps is available.

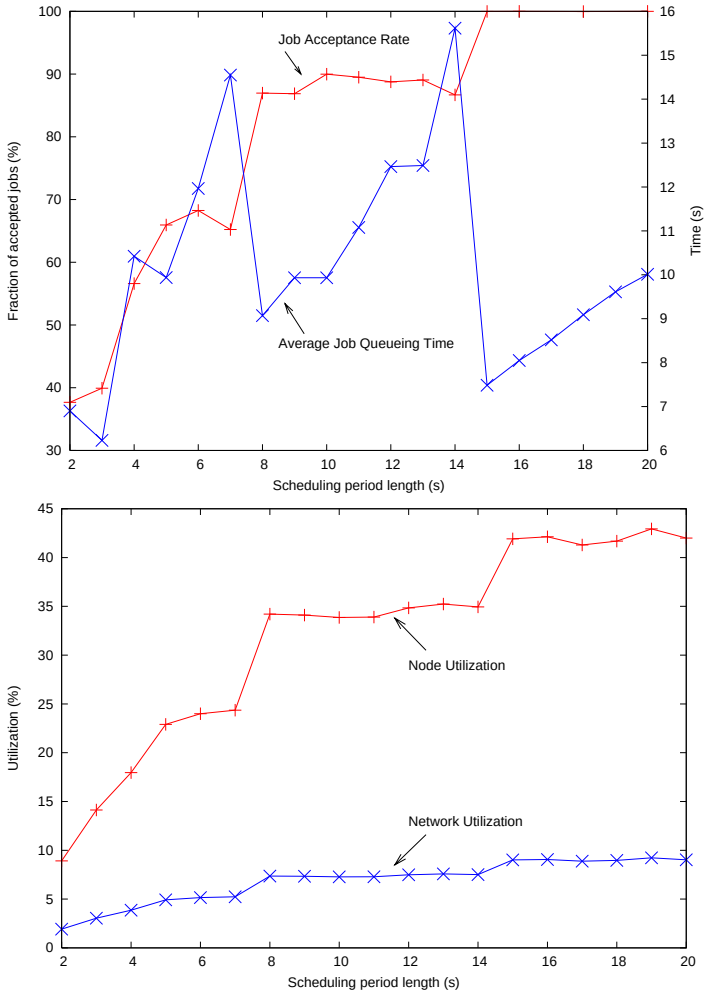


Figure 55 – Influence of period length *T*

As before, we established $(\lambda T)_{max}$ = 10. Figure 55 shows the acceptance rate for different values of the period length, in which we see a step-like function emerge. As long as the period is greater than or equal to the running time of jobs, all jobs can be accepted. Taking the period length smaller than the job running time, implies resources will be reserved for two periods, although the resources are not utilized until the end (see also Section 4.2.2.1). We see the same behavior when the period length equals whole fractions of the job running time. This also explains the spiked pattern of the average job queuing time, where optimal values are obtained by taking the period length equal to whole fractions of the job running time.

4.3 Co-allocation of Grid Resources Following the Hierarchical Scheduling Model of VIOLA

As stated in the introduction, resources in the Grid context may involve the aggregation of different compute and data resources. QoS in the context of aggregated resources means to ensure the availability of all required resources at the specified time for the certain period. Especially advanced applications benefit from the existence of different, heterogeneous resources available in Grids. For these advanced applications the required resources may also involve different sites. Resource Management of these sites is done by a local scheduling system. To allow for a coordinated allocation of resources at different sites the local scheduling systems need to support an advance reservation mechanism. In addition, a coordination entity is needed that negotiates a common timeframe for resource usage with the different local schedulers. This entity is called MetaScheduling Service.

4.3.1 Introduction to VIOLA Co-Allocation

The MetaScheduling Service (MSS) [70] developed in the VIOLA Project [71] allows the end-user to execute the individual components of his application using the most appropriate resources available. Examples of such applications are distributed multi-physics simulations where multiple resources are needed at the same time, or complex workflows where the resources are needed with some timely dependencies [72]. Additionally, having distributed applications and data, there is also a need for dedicated QoS of the network connections between the resources to support efficient execution of the applications. However, to make efficient use of the resources a reservation mechanism is needed that guarantees the availability of the selected resources including the network at the time they are needed to execute application components or a component of a workflow. Without reservation there is only a best effort approach to execute applications across multiple resources without a chance of coordination. Having reservation mechanisms allows to completely plan the execution of an application or workflow if the timely dependencies are given by the user.

Part of the VIOLA project is a UNICORE [73] based Grid testbed on top of an optical network. This testbed provides solutions to the problems addressed above: the orchestration of resources of different sites belonging to different administrative domains by the MetaScheduling Service. This service is responsible for the negotiation of agreements on resource usage with the individual local resource management systems. The agreements are made using WS-Agreement [34]. They consist of Service Level Agreements on the advance reservation of the resources needed for an application or a workflow [74]. The local resource management systems finally include the advance reservation in their individual schedules. Extending this approach to network resources as done in VIOLA allows user or application driven selection and reservation of network connections with dedicated QoS based on evolving network technologies.

4.3.2 Architecture

4.3.2.1 *The UNICORE Environment and Extension of the Client*

The Grid-system UNICORE is being developed since 1998 and has been used in various projects and production environments, mainly in Europe and Japan. UNICORE is based on a three-tier architecture, consisting of (1) a Java-Client as the user-interface to the Grid, (2) server-components at the UNICORE-sites that provide the secure access of the user to the UNICORE Grid and manage the users jobs and finally (3) the target systems which execute those jobs (see Figure 58).

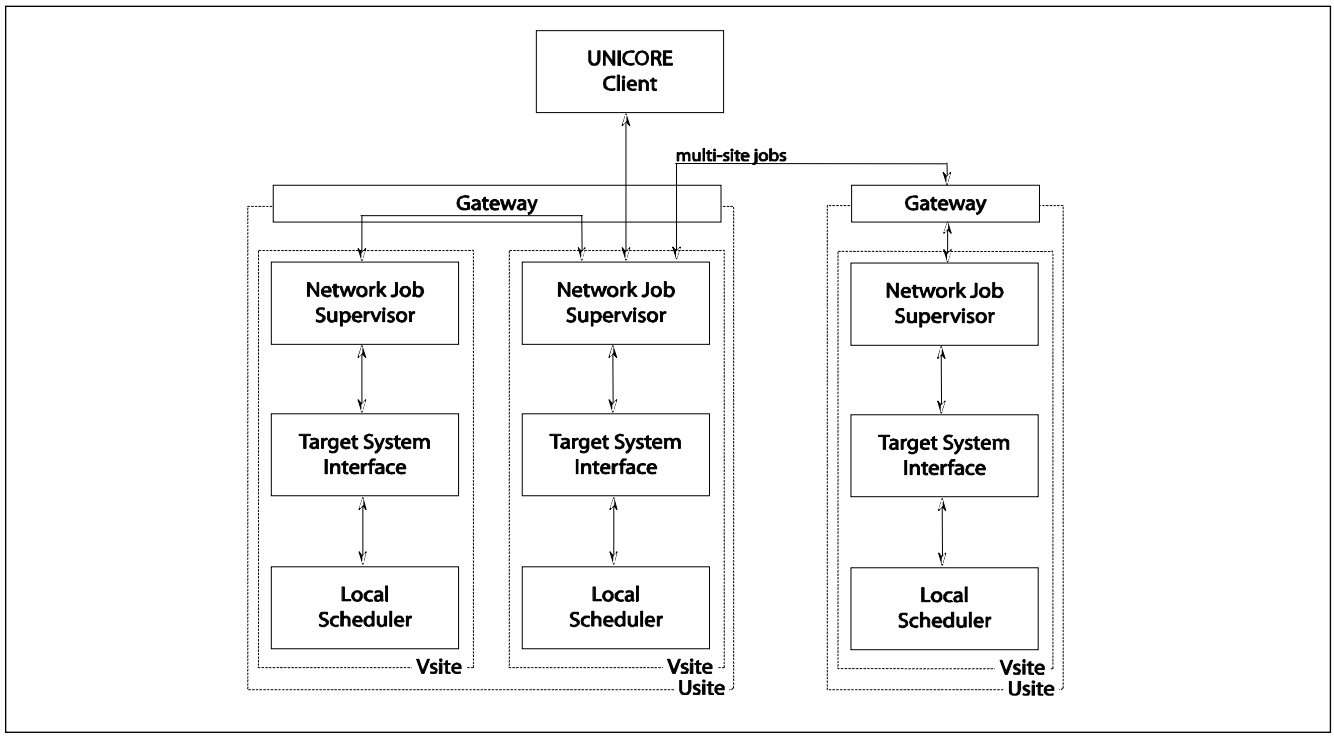


Figure 56 – Unicore Architecture.

The standard UNICORE software offers extended workflow support. UNICORE jobs are composed of subjobs that can be executed on the same or different resources (called sites). Dependencies between those subjobs can be specified, forcing them to be executed in a particular order. In addition, conditional execution and control statements allow for building loops of subjobs. However, UNICORE has no building capabilities to make advance reservations or to provide synchronous access to distributed resources. Within VIOLA, this feature has been added via a UNICORE client plugin that accesses an external MetaScheduling Service. The plugin provides a GUI that lets the user specify his job including the number of processes to run on which target systems and the required bandwidth between them. Based on this information, the client requests a reservation from the MSS. Once the reservation has been made by the MSS, it is processed like any other UNICORE job. A job may consist of a number of subjobs one for each target system that is requested. Users can retrieve output, monitor or cancel the job. In the current version, the plugin is tailored to the needs of distributed

simulations using the metacomputing-enabled MPI-implementation MetaMPICH [75]. Using it, the user not only specifies the resources needed but also further MetaMPICH-related information allowing the plugin to perform additional tasks, as e.g. distributing the different types of MetaMPICH tasks (compute tasks, network router tasks, I/O server tasks) onto the requested cluster nodes based on various policies and generating a MetaMPICH configuration file. The plugin is designed and implemented in a modular fashion, allowing easy adaptation to other types of distributed application, not based on MetaMPICH.

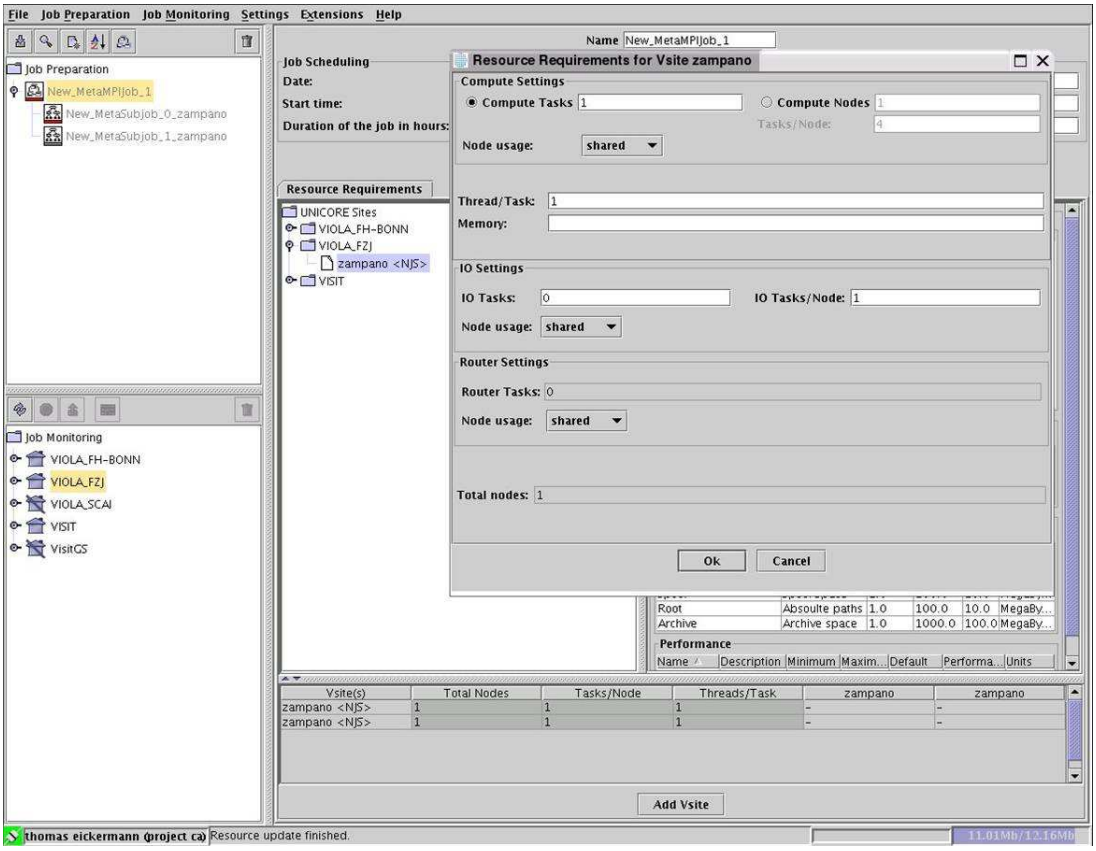


Figure 57 – Snapshot of UNICORE Client with the MetaMPICH-plugin: construction of a MetaMPICH-job.

4.3.2.2 *Meta-Scheduling Service*

Once the MSS receives the agreement proposal with the necessary information on resources and QoS needed for an application from the UNICORE client it starts to negotiate with the local Resource Management Systems (RMS) of these resources (see Figure 58). The negotiation has four main phases:

(1) querying the local RMS for free slots to execute the application within a preview period

(2) determining a common time slot

(3) if such a time-slot exists - perform a reservation request of this slot on behalf of the user
otherwise - restart the query with a later start time of the preview period

(4) check whether the reservation was made for the correct time slot on all systems,
if yes - we are done;
otherwise - restart the query with a later start time of the preview period

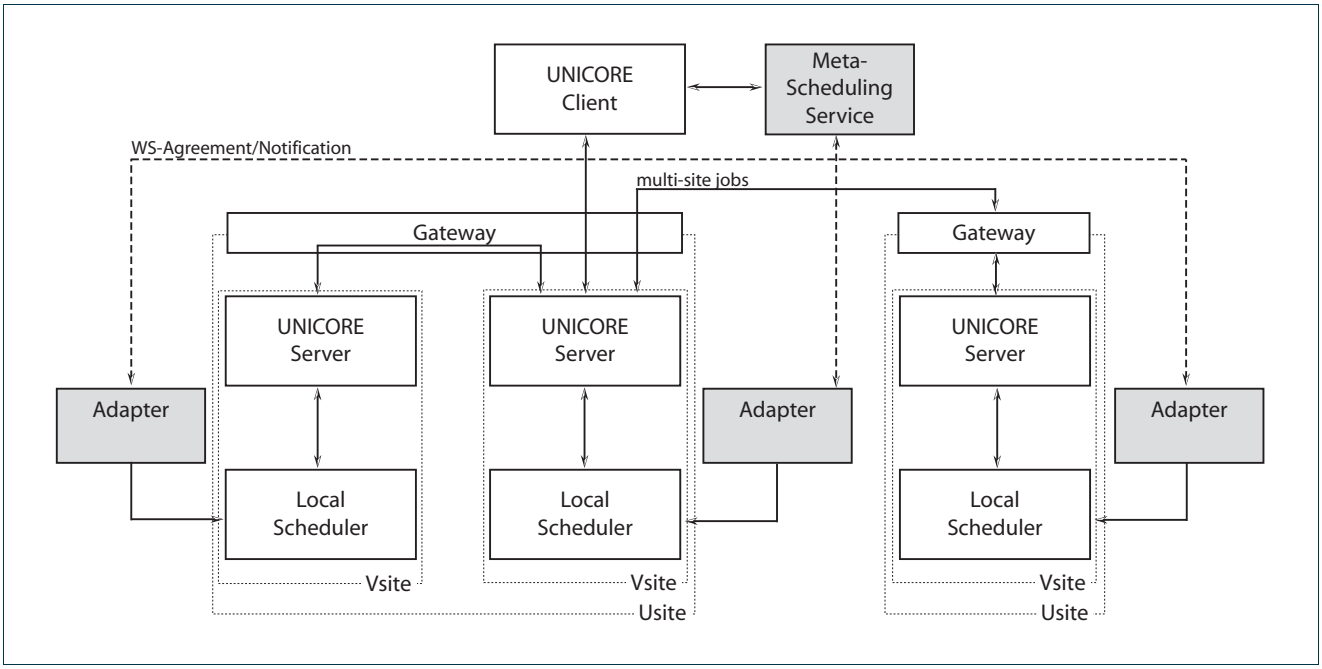


Figure 58 – Architecture of the VIOLA Meta-scheduling Environment

If no common time-slot within the local RMS's specific look-ahead times can be identified, an error is reported to the user. The pseudo-code of the co-allocation algorithm is depicted below.

```
set  n                    =  number of requested resources
set  resources[1..n]      =  requested resources
set  properties[1..n]     =  requested property per resource      # number of nodes, bandwidth,
time, ..
set  freeSlots[1..n]      =  null                                 # start time of free slots
set  endOfPreviewWindow        =  false
set  nextStartupTime      =  currentTime+someMinutes   # the starting point when looking for free slots
while (endOfPreviewWindow = false) do {
        for 1..n do in parallel {
                freeSlots[i] = ResourceAvailableAt( resources[i], properties[i], nextStartupTime)
        }
        for 1..n do {
                set needNext = false
                if ( nextStartupTime != freeSlots[i]) then {
                        if ( freeSlots[i] != null) then {
                                if( nextStartupTime < freeSlots[i]) then {
                                        set nextStartupTime =  freeSlots[i]
                                        set needNext        =  true
                                }
                        } else {
                                set endOfPreviewWindow = true}
                }
        }
}
if ( ( needNext = false ) & ( endOfPreviewWindow = false) ) then return
        freeSlots[1] else return “no commont slot found”
```

Figure 59 – MetaScheduling Algorithm.

The successful negotiation and reservation is sent back as agreement to the UNICORE client who then processes the job as usual. When the job starts at the negotiated common starting time the MSS collects the IP addresses of the participating machines (this information may not be available at an earlier time as the local scheduling system might assign the job to different nodes than planned at the time of submission) and communicates them to the network RMS which in turn is then able to manage the end-to-end connections with the requested QoS.

4.3.2.3 *Advance Reservation of Network Resources*

Taking a look at the Grid as a geographically distributed set of resources comprising computing and storage for users and their applications, the connecting network infrastructure becomes important. While sites are usually connected by IP best effort technologies, the coordination of high performance resources like meta-computing brings new requirements and challenges to the network. A site’s Internet connectivity is usually tailored to the bandwidth demands of the well-known interactive Internet applications like e-mail and web traffic. It is assumed that coupling clusters to efficiently use computing and storage resources from multiple sites requires high bandwidth (e.g. in terms of multiple Gbit/s) and low delay (e.g. as low as possible) connections with virtually exclusively usage characteristics. The idea of QoS in the network domain has been apparent for many years. In addition, the VIOLA project provides an in advance reservation interface which allows to connect sites on demand with high speed, low delay connections. These premium connectivity services can be invoked by the Meta Scheduling Service in order to provide on demand the required network QoS for multi-site jobs. The

following section presents a brief overview of the developed network reservation system ARGON (Allocation and Reservation in Grid-enabled Optic Networks) including the advance reservation capable interface for the Grid application layer offering connectivity services with a specified QoS on top of the optical network between the Grid sites in the VIOLA network. This allows for a combined reservation of cluster and network resources and to achieve high network utilization. Figure 60 shows the north- and southbound interfaces of ARGON: The Network Resource Management (NRM) service in the VIOLA testbed.

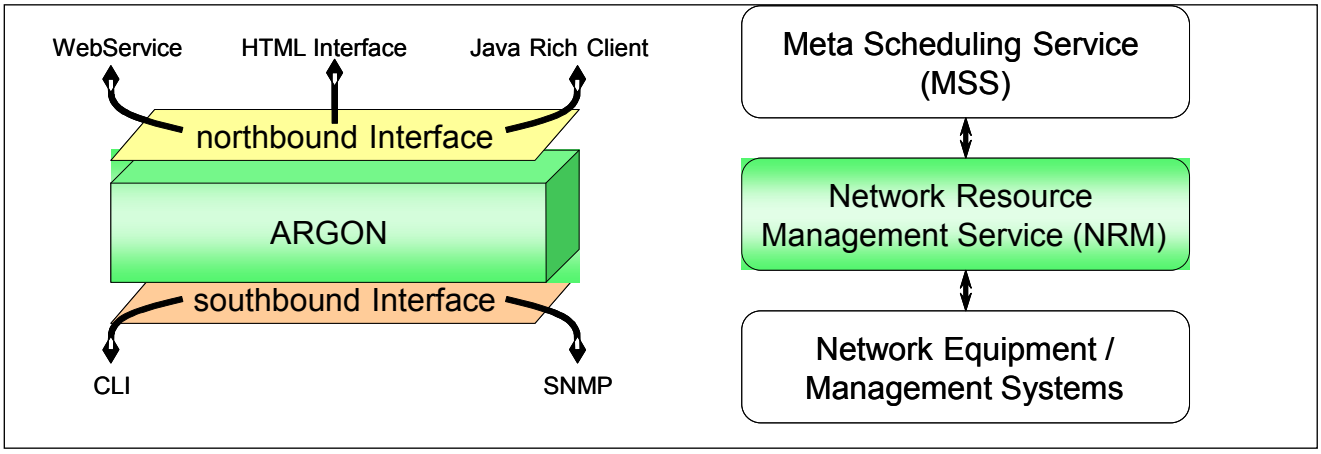


Figure 60 – North- and southbound Interfaces of ARGON

ARGON is designed to provide a set of network related services to the Grid community, e.g. serve advance reservation requests by the upper layer (e.g. MSS). Services include the instantaneous setup of network connections if the requested resources are available for the specified duration of time. At this level ARGON tries to hide the details of the network technologies, i.e. the user or application have to specify only the QoS requirements for a service and the service endpoints. ARGON maps abstract premium connectivity services to specific layer 2 and layer 3 network services via MPLS as well as point-to-point connectivity services via GMPLS. Beside the details of a single service, a set of services can be bundled in a single request for reservation. Hence, a reservation may consist of several services with chronological dependencies which may themselves consist of several connections as a basis for the service. Consequently, the whole reservation can be regarded as a transaction: All services contained are accepted, rejected or postponed as a whole. This also applies for malleable reservations where the overall service of the reservation can be stretched or compressed in the same way. The idea of malleable reservations is sketched in Figure 61: a data amount has to be transferred, and according to the present resource allocation and reservation parameters ARGON can choose an appropriate duration and capacity frame to schedule the service.

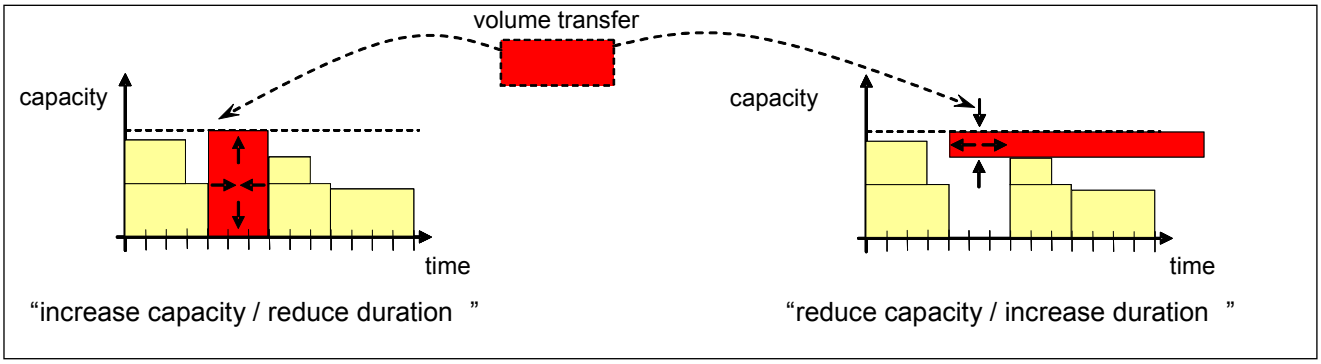


Figure 61: Malleable Reservations.

In order to allow for automated resource coordination and provisioning, the northbound interface is implemented as a Web Service and accessible via SOAP. It allows for the Meta Scheduling Service (MSS) or other applications to reserve network capacity and scheduled services. The interface currently consists of five message types for: (i) the reservation of resources, (ii) the cancellation of reservations, (iii) the query of reservation related information, (iv) the query of availability information and (v) the binding of additional information for provisioning purposes. Availability information and binding of provisioning information are especially important for the co-allocation of resources via the MSS. The availability request helps to find a common time slot for cluster and network resources. A late binding of provisioning information allows for the MSS and the local scheduling systems respectively to appoint the cluster nodes used for a reservation just in time before the provisioning. Thus, at the time of reservation only the service endpoint (e.g. provider or consumer edge router), but not the identity of the cluster nodes needs to be known. The provisioning information may consist of ports of the router to which the cluster nodes are attached and/or IP addresses.

The southbound interface of ARGON to the network components uses standard network management protocols – if available – to initiate MPLS/GMPLS based signaling to control both the MPLS and the GMPLS domain. At the time of writing, the primary interfaces to the network equipment use either a Command Line Interface (CLI) – which is not only vendor specific but also version dependent – and SNMP if possible. It is also envisioned to integrate vendor specific management interfaces that support XML message transfer with a higher layer of abstraction. In the context of MPLS two services are favored by ARGON: A layer 3 based tunnel service and VPLS. The layer 3 based tunnel service utilizes MPLS traffic engineered point-to-point tunnels which convey IP packets. The Virtual Private LAN Service connects sites by means of layer 2 connectivity, i.e. bridging of LANs and VLANs across multiple sites. Beside these technologies, also H-VPLS and virtual private routed network solutions are under investigation. In the GMPLS domain SDH equipment is used in the VIOLA testbed. This equipment is configured via CLI or vendor specific UNI client proxies which trigger RSVP-based UNI signaling to establish SDH point-to-point connections. Beside the signaling and provisioning of network resources, ARGON uses the southbound interface to gather information about the networks, e.g. topological information is extracted from the OSPF database of the MPLS and GMPLS control plane.

The core of ARGON utilizes the network topology information to compute the possible paths in the network and thus to realize and plan the requested services in advance. Although the network equipment in the VIOLA testbed allows for traffic engineering, the on demand and in advance reservations must be handled by ARGON. Protocols used for traffic engineering – like OSPF-TE and RSVP-TE – provide means for instantaneous path computation and signaling within the network components. Pre-planning of future capacity usage is therefore

left to the core of ARGON which supervises the resource usage in the underlying network layers (MPLS and GMPLS).

4.3.3 Implications for the Phosphorus project

The MetaScheduling Service developed in the VIOLA project will be enhanced to cope with the requirements of the applications in the Phosphorus project. A detailed discussion on these requirements and the implications for the MetaScheduling Service can be found in [77]. The interface for advance reservation of network resources as described in Section 4.3.2.3 has strongly influenced the development of the interface for the network service plane which is specified in [76]. A detailed discussion about advance reservation can be found in [78].

4.4 Conclusions

In Section 4 we formulate different problems of co-allocation of resources in a Grid environment and propose solutions to these problems. More specifically, we assume that the submitted task can consist of a number of other “subtasks” with various interdependencies. Thus the user request a co-allocation of (possible different) resources and service guarantees over the “execution” of the subtasks on that resources.

In Section 4.1 we examined the “Anycast” problem, presented two variations to this problem and two algorithms to address these variations.

In the first case we examined the concurrent co-allocation of the communication and computation resources. The proposed algorithm, called SAMCRA, is a multi-constrained routing and scheduling algorithm, which uses the path delay and the computation (cluster) load as selection metrics. This algorithm takes into account the path delay from source to destination and the servers’ current loads. The simulation results showed that both the source based and hop-by-hop application of SAMCRA provide a reliable job request distribution over the available resources. We introduced a new sub-path evaluation ordering method for the SAMCRA algorithm. Contrary to an ordering based on the q-norm length definition, this novel ordering guarantees that SAMCRA finds the exact solution for the multi-constrained optimal path (MCOP) problem. The new path evaluation ordering applies both to unicast and anycast routing problems. Results showed that both the centralized and hop-by-hop variant of the updated SAMCRA algorithm approach the acceptance probability of the maximum flow upper boundary, which proves the practical optimality of the proposed approach.

In the second formulation of the anycast problem we assumed that task processing consists of two successive steps: (i) the transfer of data from a site to the computation resource and (ii) the execution of the task at the computation resource, forming in this way a simple workflow. We presented a multicost algorithm for the joint selection of the communication and computation resources to be used by a task. 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.

In Section 4.2 we addressed a problem of concurrent co-allocation of network, storage and computation resources. More specifically, each job is characterized by its processing rate, the size of the input (output) datasets, and the input (output) bandwidth. Scheduling is performed at periodic instants. We presented an ILP formulation which tells us whether or not a job is accepted for execution, and if so, which resources it may use. Accepted jobs are all started immediately and at the same time; rejected jobs are transferred automatically to the next scheduling round (i.e. a job will occupy its assigned resources during a whole number of period lengths). The results show that the scheduler behaves as expected under a broad range of job request loads. Still, even when much of the Grid's resources are used, the scheduler optimally places new jobs. We also investigated the effect of available bandwidth in the network and determined a cut-off value; staying above this value will always yield acceptable performance. Finally, we studied the effects of the scheduling period length, and clearly showed the existence of optimal values.

In Section 4.3 we presented the MetaScheduling Service (MSS) developed in the VIOLA. The MSS of VIOLA can tackle complicated workflows allowing the end-user to execute the individual components of his application using the most appropriate resources available. The MSS is able to orchestrate resources of different sites belonging to different administrative domains, while it is responsible for the negotiation of agreements on resource usage with the individual local resource management systems. The MMS developed in the VIOLA project will be enhanced to cope with the requirements of the applications in the Phosphorus project. Finally, we give a brief overview of the network reservation system ARGON. ARGON includes an advance reservation capable interface for the Grid application layer and thus offers connectivity services with a specified QoS on top of the optical network.

4.5 Symbols

Symbol	Meaning
j	A task or a job. In this deliverable, the words “task” and “job” are used interchangeable
l^r_j	Input data size of job j
l^w_j	Output data size of job j
b^r_j	Input bandwidth of job j
b^w_j	Output bandwidth of job j

b_l	Duration of input data (duration of the input burst)
w_j	The number of CPUs in which job *j* requests to be executed
h_j	Computational workload of job *j* on a single CPU (in MI)
b_m	Duration of the job execution on a machine *m*
c_j	The processing rate that job *j* request to be served by a computation resource
q_j	Budget of job *j*
Q_j	Cost of job *j*
S	The source node (can be a scheduler or a data repository site)
L	Set of links
l	A link
d_l	The delay of link *l*
C_l	Bandwidth of link *l*
$\hat{C}_l$	Binary capacity availability vector of link *l*. Also abbreviated as CAV.
τ_l	The discretization step of the capacity availability profile. The timeslot used in $\hat{C}_l$
d^l	The maximum size of $\hat{C}_l$
Q_l	Cost of using link *l*
M	The set of computation resources
m	A computation resource (usually a cluster of CPUs)
W_m	Number of CPUs of cluster *m*

H_m	Processing capacity of one CPU of cluster *m.*
C_m	Processing capacity of computation resource *m.* If we have a cluster then $C_m = W_m \cdot H_m$
$\hat{W}_m(w)$	Binary w-cluster availability vector of cluster m. Abbreviated as MAV.
τ_m	The discretization step of the r-cluster availability profile. The timeslot used in $\hat{W}_m(r)$
d^m	The maximum size of $\hat{W}_m(r)$
Q_m	Cost of using computation resource *m*
R	Data repository site or storage resource
D_r	Storage capacity of storage resource *r*
Q_r	Cost of using storage resource *r*
TO_{path}	The time to start transmission over the path
$TO_{cluster}$	The time to start execution of the task at the cluster
$\text{LSH}_x(\hat{C}_l)$	Left shift of vector $\hat{C}_l$ by *x* elements
$\text{MUV}_x(\hat{W}_m)$	Set zeros in the first *x* elements of vector $\hat{W}_m$
p	A path
$R_p(b)$ on $\hat{C}_p$	The first position after which $\hat{C}_p$ has *b* consecutive ones. Equally, the earliest time after which a burst of duration *b* can start its transmission on path *p*
EST*(p,b)*	The earliest time that the task can reach cluster *m*, defined as the end of path *p*
$\hat{W}_m(p,b)$,	The time periods that *S* (start of *p*) can schedule the task over path *p* at cluster *m (end of p)*
V_l	The cost vector of link *l*
$V(p)$	The cost vector of path *p*

pm	A (path, cluster) pair *pm* of a path *p* ending to a cluster *m*
V(*pm*)	The cost vector of *pm* pair
$P_{n\text{-}d}$	The set of non-dominated paths between the source and all the network nodes
$PM_{n\text{-}d}$	The set of candidate non-dominated (path, cluster) pairs from source to all the clusters that can process the task

Table 19: Symbols for Section 4

5 Conclusions

Grid Networks offer a transparent interface to geographically scattered communication, computation, storage and other resources. The Phosphorus project emphasis is on network-related aspects of Grids, and thus its focus is on the applications that are heavily dependent on communication resources. Many of these applications however have also significant computational dependencies.

Resource management and scheduling is a key to the success of Grid Networks, since it determines the efficiency with which resources are used and the Quality of Service (QoS) provided to the users. In this deliverable we propose and evaluate QoS-aware and fair scheduling algorithms for Grid Networks. These algorithms are capable of optimally mapping tasks to resources, considering the task characteristics and the QoS requirements. Although we mainly address scheduling problems on computation resources, we also look at joint scheduling of communication and computation resources problems, which are more related to the Phosphorus project.

In Section 1 we presented the objectives and introductory issues of this deliverable. The users, applications or Virtual Organizations (VOs), have a number of QoS requirements, most important of which are the end-to-end delay of the tasks, the communication bandwidth used, and the computational capacity allocated to the tasks. The applications designed to run in the Phosphorus testbed produce tasks that have none, one or many of these QoS requirements (service guarantees). Tasks (also applications or users) with one or more hard or strict requirements are referred to as Guaranteed Service (GS) tasks (applications or users correspondingly), while tasks with no requirements are referred to as Best Effort (BE) tasks. Our proposed algorithms are designed to serve both kinds of tasks.

Since the sharing of resources is the “raison d’ etre” of Grid Networks, fairness is a concept that is inherent in scheduling in Grids. 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 scheduling algorithms proposed are centralized, consist of two-phases (“task-ordering” and “task-to-resource assignment” phase) and incorporate fairness considerations in one of both of these phases. The algorithms investigated, use the Max-Min fair definition of sharing. The algorithms assume that the processors are time-shared and provide fairness on a per user basis or on a per task basis through the Weight Fair Queueing algorithm. 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, by better

exploiting the available resources, and also improve the performance by providing better QoS to the users.

The “best effort” service is inadequate for GS tasks (or users) requiring QoS guarantees. In Grid Networks, such a requirement is in most cases the total time it takes for a task to be completed. 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 (or tasks), with varying priorities. In our simulation experiments, data from a real Grid Network were used, validating in this way the appropriateness and usefulness of the proposed framework. Our simulation study 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. Finally, various other useful features were investigated in the context of our QoS framework, such as scheduling without a-priori knowledge of task workloads and task migration.

The joint scheduling of communication, computation, storage and other resources is another very important topic in Grids. The corresponding scheduling algorithms we proposed consider the satisfaction of two or more QoS requirements, by co-allocating resources either concurrently or successively with various interdependencies. In Section 4 a multi-constrained routing and scheduling algorithm was presented, where the communication and the computation resources were concurrently co-allocated (allocated in the same time-frame). The results showed the practical optimality of the proposed approach. To address the problem of successive scheduling of communication and computation resources we also proposed a multicost algorithm that uses advance reservations on the corresponding resources. We presented an optimal non-polynomial and a heuristic polynomial algorithm. The results of our experiments indicate that important performance benefits can be obtained by jointly optimizing the use of the communication and computation resources. We proved that the heuristic approach combines the strengths of the optimal multicost algorithm with a low computation complexity. In the same section we also addressed the problem of the concurrent co-allocation of communication, computation and storage resources. Specifically, we present an ILP formulation which incorporates two functions: task admission control and resource assignment. A solution of the ILP model tells us if a task is accepted for execution, and if so, which resources it may use. The simulation results showed that the scheduler behaves efficiently under a broad range of conditions (task request loads, resources usage, available bandwidth, etc). We also presented a real MetaScheduling Service (MSS), developed in the context of the VIOLA project. The MSS is responsible for the orchestration of the distributed resources, the selection of the most appropriate available resources and the negotiation of the resource usage. Finally, we gave a brief overview of an advance communication reservation system, called ARGON, developed in the VIOLA project. Argon is one of the Network Resource Provision Systems (NRPS) that inter-operates with Phosphorus reservation system.

6 References

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

[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) - Volume 00 (September 20 - 24, 2004). ENC. IEEE Computer Society, Washington, DC, 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. IEEE Press, New York, USA, March 2005, 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). IEEE Computer Society, 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 (July 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, Germany, May 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 Proc. of IWQOS, UK, 1999, pp. 27-36.

[12] S. N. Bhatti, S.-A. S¸rensen, 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] I. Foster, and D. Gannon, “The Open Grid Services Architecture Platform”, Global Grid Forum Drafts, http://www.ggf.org/ogsa-wg, 2003.

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

[15] 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.

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

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

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

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

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

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

[22] 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 Trans. on Systems Man and Cybernetics, Part-B, Vol. 34, No. 2, pp. 1235-1247, April 2004.

[23] S. Keshav, An Engineering Approach to Computer Networking. Addison-Wesley, 1997.

[24] 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.

[25] B. Briscoe, “Flow Rate Fairness: Dismantling a Religion”, In SIGCOMM Comput. Commun. Rev., 2007, Vol. 37, No. 2, pp. 63-74.

[26] G. Lanfermann, G. Allen, T. Radke and E. and Seidel, “Nomadic Migration: A New Tool for Dynamic Grid Computing”, In Proc. of HPDC, United States, 2001.

[27] S. Frechette and D. R. Avresky, “Method for Task Migration in Grid Environments, In Proceedings of the NCA”, United States, 2005, Vol. 00, pp. 49-58.

[28] K. Sato, A. Fukuda, T. Koita, Y. Inoue, “Heuristic Scheduling and Process Migration on the Grid”, In Proc. of GCA, United States, 2006, pp. 68-74.

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

[30] 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), Vol. 14, No. 13-15, pp. 1175-1220, 2002.

[31] R. Raman, M. Livny, and M. H. Solomon, “Policy driven heterogeneous resource co-allocation with gangmatching,” In Proc. of the 12th IEEE Int’l Symp. on High Performance Distributed Computing (HPDC-12), 2000, pp. 80–89..

[32] J. Li, R. Yahyapour, “On Advantages of Grid Computing for Parallel Job Scheduling,” in Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID2002). Berlin: IEEE Press, May 2002, pp. 39–46.

[33] K. Czajkowski, I. T. Foster, and C. Kesselman, “Resource co-allocation in computational grids,” in Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing (HPDC-8), pp. 219-228, 1999., 1999.

[34] 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)”, March 2007. 15 Mar 2007 <https://forge.Gridforum.org/sf/docman/do/downloadDocument/projects.graapwg/docman.root.current drafts/doc6091.

[35] T. Roeblitz and A. Reinefeld, “Co-reservation with the concept of virtual resources,” in Proceedings of the 5th IEEE International Symposium on Cluster Computing and the Grid, Cardiff, Wales, UK, May 9-12. IEEE Computer Society, 2005.

[36] H. Mohamed and D. Epema. “The design and implementation of the koala co-allocating grid scheduler”. In P. Sloot, A. Hoekstra, T. Priol, A. Reinefeld, and M. Bubak, editors, Advances in Grid Computing - EGC 2005, volume 3470 of Lecture Notes in Computer Science, pages 640–650, Amsterdam, February 2005. Springer.

[37] J. Yu and R. Buyya, “A Taxonomy of Scientific Workflow Systems for Grid Computing”, Special Issue on Scientific Workflows, SIGMOD Record, Volume 34, Number 3, Pages: 44-49, 2005.

[38] 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.

[39] Deelman, J. Blythe, Y. Gil, C. Kesselman, G. Mehta ,K. Vahi, K. Blackburn , A. Lazzarini, A. Arbree, R. Cavanaugh, S. Koranda, “Mapping Abstract Complex Workflows onto Grid Environments”, Journal of Grid Computing, Vol.1:25-39, 2003.

[40] J. Cao, S. A. Jarvis, S. Saini, G. R. Nudd, “GridFlow:Workflow Management for Grid Computing”, In 3rd International Symposium on Cluster Computing and the Grid (CCGrid), Tokyo, Japan, 2003.

[41] 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, 2004; 19-36.

[42] T. Fahringer et al, “ASKALON: a tool set for cluster and Grid computing”, Concurrency and Computation: Practice and Experience, 17:143-169, Wiley InterScience, 2005.

[43] S. McGough, L. Young, A. Afzal, S. Newhouse, and J. Darlington, “Workflow Enactment in ICENI”, UK e-Science All Hands Meeting, Nottingham, UK, Sep. 2004; 894-900.

[44] G. Hoo, W. Johnston, I. Foster, A. Roy, “QoS as Middleware: Bandwidth Reservation System Design”, HPDC, pp. 345-346, 1999.

[45] K. Ranganathan, I. Foster. “Decoupling computation and data scheduling in distributed data-intensive applications”, HPDC, 2002.

[46] S. Venugopal, R. Buyya, L. Winton, “A Grid Service Broker for Scheduling Distributed Data-Oriented Applications on Global Grids”, MGC 2004.

[47] K. Nahrstedt, H. Chu, S. Narayan, “QoS-aware resource management for distributed multimedia applications”, J. High Speed Networks, pp. 229-257.

[48] R. Guerin, A. Orda, “Networks with Advance Reservations: The Routing Perspective”, Infocom, 2000.

[49] Job Submission Description Language specification: www.ogf.org/documents/GFD.56.pdf.

[50] E. Varvarigos, V. Sourlas, K. Christodoulopoulos, “Routing and Scheduling Connections with Advance Reservations”, submitted to Computer Networks.

[51] F. Gutierrez, E. Varvarigos, S. Vassiliadis, "Multicost Routing in Max-Min Fair Networks", Allerton Conference, 2000.

[52] K. Manousakis, V. Sourlas, K. Christodoulopoulos, E. Varvarigos, K. Vlachos, “A Bandwidth monitoring mechanism: Enhancing SNMP to record Timed Resource Reservations”, J. Network and Systems Management, pp. 583-597, 2006.

[53] The Network Simulator (ns2): www.isi.udu/nsnam/ns.

[54] J. Turner, “Terabit burst switching,” Journal of High Speed Networks, Vol. 8, No. 1, pp. 3-16, 1999.

[55] Y. Xiong, M. Vandenhoute, and H.C. Cankaya, “Control Architecture in Optical burst-switched WDM Networks,” IEEE Journal on Selected Areas in Communications, Vol. 18, pp. 1838–1851, 2000.

[56] C. Qiao, M. Yoo, “Optical burst switching (OBS)–a new paradigm for an optical Internet,” J. High Speed Networks, vol. 8, no. 1, pp. 69–84, 1999.

[57] Grid Optical Burst Switched Networks: www.ogf.org/Public_Comment_Docs/Documents/Jan-2007/OGF_GHPN_GOBS_final.pdf

[58] E. Varvarigos, V. Sharma, “An efficient reservation connection control protocol for gigabit networks”, J. Computer Networks and ISDN Systems 30, pp. 1135–1156, 1998.

[59] P. van Mieghem, and F. Kuipers, “Concepts of Exact QoS Routing Algorithms”, IEEE/ACM Transactions on Networking, 12(5):851-864, Oct 2004

[60] D. Xuan, W. Jia, W. Zhao, and H. Zhu, “A Routing Protocol for Anycast Messages,” IEEE Transactions on Parallel and Distributed Systems, 11(6):571–588, June 2000

[61] F. Kuipers, and P. Van Mieghem, “Conditions That Impact the Complexity of Qos Routing,” IEEE/ACM Transactions on Networking, 13(4):717–730, Sep 2005

[62] P. Van Mieghem, H. De Neve, and F. Kuipers, “Hop-by-Hop Quality of Service Routing,” Elsevier Computer Networks, 37(3–4):407–423, Nov 2001

[63] I.S. MacKenzie, and C. Ware, “Lag as a Determinant of Human Performance in Interactive Systems,” in Proceedings of the ACM Conference on Human Factors in Computing Systems – InterCHI’93, Amsterdam, The Netherlands, Apr 1993, pp. 488–493.

[64] S. De Maesschalck, D. Colle, I. Lievens, M. Pickavet, P. Demeester, C. Mauz, M. Jaeger, R. Inkret, B. Mikac, and J. Derkacz, “Pan-European Optical Transport Networks: An Availability-based Comparison,” Photonic Network Communications, 5(3):203–225, May 2003

[65] W. Feng, F. Chang, W. Feng, and J. Walpole, “A Traffic Characterization of Popular On-Line Games,” IEEE/ACM Transactions on Networking, 13(3):488–500, June 2005

[66] G.L. Nemhauser, L.A. Wolsey, “Integer and Combinatorial Optimization", John Wiley & Sons, 1988

[67] H.A. James, K.A. Hawick, and P.D. Coddington, “Scheduling Independent Tasks on Metacomputing Systems", Proc. of Parallel and Distributed Computing Systems, Aug 1999

[68] V. Hamscher, U. Schwiegelshohn, A. Streit, and R. Yahyapour, “Evaluation of Job-Scheduling Strategies for Grid Computing", Proc. of High Performance Computing, 2000

[69] K. Ranganathan, and I. Foster, “Simulation Studies of Computation and Data Scheduling Algorithms for Data Grids", Journal of Grid Computing, 1:53-62, 2003

[70] O. Wäldrich, Ph.Wieder, and W. Ziegler, “A Meta-scheduling Service for Coallocating Arbitrary Types of Resources”, In Proc. of the Second Grid Resource Management Workshop (GRMWS’05) in conjunction with the Sixth International Conference on Parallel Processing and Applied Mathematics (PPAM2005), volume 3911 of Lecture Notes in Computer Science, pages 782–791, Poznan, Poland, September 11–14, 2006, Springer.

[71] VIOLA – Vertically Integrated Optical Testbed for Large Application in DFN, Mar 2007 http://www.viola-testbed.de/

[72] G. Quecke and W. Ziegler, “MeSch – An Approach to Resource Management in a Distributed Environment”, In Proc. of 1st IEEE/ACM International Workshop on Grid Computing (Grid 2000), volume 1971 of Lecture Notes in Computer Science, pages 47–54, Springer, 2000.

[73] 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” In L. Grandinetti, editor, Grid Computing: The New Frontiers of High Performance Processing, Advances in Parallel Computing 14, Elsevier, 2005.

[74] Ph. Wieder, O. Wäldrich, R. Yahyapour, and W. Ziegler, “Improving Workflow execution through SLA-based Advance Reservation”, In Intregrated Research in Grid Computing, CoreGRID Integration Workshop, pages 333 – 344, Krakow, Poland, 2006, ISBN: 83-915141-6-1.

[75] B. Bierbaum, C. Clauss, Th. Eickermann, L. Kirtchakova, A. Krechel, S. Springstubbe, O. Wäldrich, Ph.Wieder, and W. Ziegler, “Reliable Orchestration of distributed MPI-Applications in a UNICORE-based Grid with MetaMPICH and MetaScheduling”, In Proc. of the EuroPVM/MPI 2006, volume 4192 of Lecture Notes in Computer Science, pages 174 – 183, Springer, 2006.

[76] Phosphorus-WP1-D.1.1, “Requirements and specification of the interfaces and architecture for interoperability between NRPS, GMPLS, Middleware”.

[77] Phosphorus-WP3-D3.1, “Use-cases, requirements and design of changes and extensions of the applications and middleware”.

[78] Phosphorus-WP5-D5.4, “Support for advance reservations in Scheduling”.

[79] Phosphorus-WP5-D5.3, “Grid Routing Algorithms”.

[80] P. Kokkinos, E. A. 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.

[81] P. Kokkinos, E. A. Varvarigos, N. Doulamis, “A Framework for Providing Hard Delay Guarantees in Grid Computing”, accepted in e-Science 2007.

[82] P. Kokkinos, E. A. Varvarigos, “A Framework for Providing Hard Delay Guarantees and User Fairness in Grid Computing”, submitted in IEEE Transactions on Parallel and Distributed Systems.

[83] H. Dafouli, P. Kokkinos, E. A. Varvarigos, “Fair Completion Time Estimation Scheduling in Grid Networks”, submitted to ICC 2008.

7 Acronyms

Acronym	Interpretation
AFTO	Adjusted Fair Task Order
AW	Availability Weight
AWMC-T	AW heuristic multicost algorithm for the joint communication and computation task scheduling
BE	Best Effort
CAV	Capacity Availability Vector
CDF	Cumulative Distribution Function
ECT	Earliest Completion Time
EDF	Earliest Deadline First
EST	Earliest Starting Time
FCFS	First Come First Serve
FCTE	Fair Completion Time Estimation
FIFO	First In First Out
FUTS	Fair User and Task Scheduling

GBE	GS_BE_EQUAL
GBP	GS_BE_PRIORITY
GOBS	Grid Optical Bursts Switched
GPS	Generalized Processor Sharing
GS	Guaranteed Service
ILP	Integer Linear Programming
LLF	Least Length First
MC-B	Optimal multicost burst routing and scheduling algorithm
MCOP	Multi-constrained optimal path
MC-T	Optimal multicost algorithm for the joint communication and computation task scheduling
MI	Million Instructions
MIPS	Millions Instructions Per Second
MMFS	Max-min Fair Share
MMS	MetaScheduling Service
OBS	Optical Burst Switching
OGF	Open Grid Forum

PE	Processing Elements
QoS	Quality of Service
RTT	Round-Trip Tip
SAMCRA	Self Adaptive Multiple Constraints Routing Algorithm
SFTO	Simple Fair Task Order
TO	Time Offsets
UI	User Interface
VO	Virtual Organization
WFQ	Weighted Fair Queuing

Appendix A Ideal Non-Weighted Max-Min Fair Sharing Algorithm

The non-weighted Max-Min fair sharing algorithm is described as follows. The demanded computation rates X_i, *i*=1,2,…,*N,* of the tasks are sorted in ascending order, say, $X_1<X_2<\bullet\bullet\bullet<X_N$. Initially, we assign capacity *C*/*N* to the task T_1 with the smallest demand X_1, where *C* is the total Grid computation capacity (Eq. (1)). If the fair share *C*/*N* is more than the demanded rate X_1 of task T_1, the unused excess capacity of $C/N - X_1$ is again equally shared to the remaining tasks *N*-1 so that each of them gets additional capacity $(C/N + (C/N - X_1))/(N-1)$. This may be larger than what task T_2 needs, in which case the excess capacity is again equally shared among the remaining *N*-2 tasks, and this process continues until there is no computation capacity left to distribute or until all tasks have been assigned capacity equal to their demanded computation rates. When the process terminates each task has been assigned no more capacity than what it needs, and, if its demand was not satisfied, no less capacity than what any other task with a greater demand has been assigned. This scheme is called non-weighted max-min fair sharing since it maximizes the minimum share of a task whose demanded computation rate is not fully satisfied.

We can mathematically describe the previous algorithm as follows. We denote by $r_i(n)$ the *non adjusted fair computation rate* of the task T_i at the n[th] iteration of the algorithm. Then $r_i(n)$ is given by

$$r_i(n) = \begin{cases} X_i & if\ X_i < \sum_{k=0}^{n} O(k) \\ \sum_{k=0}^{n} O(k) & if\ X_i \geq \sum_{k=0}^{n} O(k) \end{cases},\ n \geq 0 \qquad \text{(A1a)}$$

where

$$O(n) = \frac{C - \sum_{i=1}^{N} r_i(n-1)}{card\{N(n)\}},\ n \geq 1 \qquad \text{(A2b)}$$

with

$$O(0) = C/N \qquad \text{(A1c)}$$

In Eq. (A1b), $N(n)$ is the set of tasks whose assigned fair rates are smaller than their demanded computation rates at the beginning of the n[th] iteration, that is,

$$N(n) = \{T_i : X_i > r_i(n-1)\} \text{ and } N(0)=N, \quad \text{(A2)}$$

while the function $card(\cdot)$ returns the cardinality of a set. The process is terminated at the first iteration n_o at which either $O(n_o) = 0$ or the number $card\{N(n_o)\}$ =0. The former case indicates congestion, while the latter indicates that the total Grid computation capacity can satisfy all the demanded task rates, that is,

$$\sum_{i=1}^{N} X_i < C \quad \text{(A3)}$$

The non-adjusted fair computation rate r_i of task T_i are obtained at the end of the process as

$$r_i = r_i(n_0) \quad \text{(A4)}$$

Appendix B Ideal Weighted Max-Min Fair Sharing Algorithm

In this case, we allocate computation capacity as if the number of submitted tasks is equal to the sum of the respective weights, that is, as if there were $\tilde{N} = \sum_{i=1}^{N} \varphi_i$ virtual tasks. An equal fair sharing is performed for all $\tilde{N}$ virtual tasks using the algorithm of Section 2.1.4 Eq. (A1) is then modified as follows

$$r_i(n) = \begin{cases} X_i & if\ X_i < \varphi_i \cdot \sum_{k=0}^{n} O(k) \\ \varphi_i \cdot \sum_{k=0}^{n} O(k) & if\ X_i \geq \varphi_i \cdot \sum_{k=0}^{n} O(k) \end{cases}, \ n \geq 0 \qquad \text{(B1)}$$

where

$$O(n) = \frac{C - \sum_{i=1}^{N} r_i(n-1)}{\tilde{N}(n)}, \ n \geq 1 \qquad \text{(B2)}$$

and $$O(0) = C / \tilde{N}, \text{ with } \tilde{N} = \sum_{i=1}^{N} \varphi_i \qquad \text{(B3)}$$

$\tilde{N}(n)$ is the sum of the weights of the tasks whose assigned fair rates are smaller than their demanded computation rates at the beginning of the n[th] iteration of the algorithm, that is:

$$\tilde{N}(n) = \sum_{i} \varphi_i : \text{for all } i : X_i > r_i(n-1) \text{ and } \tilde{N}(0) = \tilde{N} \qquad \text{(B4)}$$

The process is terminated at an iteration n_o at which either $O(n_o) = 0$ or $card\{\tilde{N}(n_o)\}$ =0.

Appendix C Max-Min Fair Scheduling

Since tasks are non-preemptable (they cannot be split in smaller units that are executed on different processors), the sum of the rates of the tasks assigned for execution to a processor may be smaller than the processor capacity, and some processors may not be fully utilized. A processor with unused capacity will be called an *underflow* processor. In an optimal solution, tasks assigned to underflow processors have schedulable rates that are equal to their respective fair rates, $r_i^s = r_i$, and do not contribute to the error *E* (otherwise we could assign additional capacity to those tasks and reduce the error). Only tasks assigned to fully utilized processors may contribute to the error *E* (but not all tasks assigned to fully utilized processors contribute to the error).

We define the overflow O_j of processor *j* as

$$O_j = \max\{0, \sum_{i \in P_j} r_i - c_j\} \quad \text{(C1a)}$$

and the underflow U_k of processor *k* as

$$U_k = \min\{0, \sum_{i \in P_k} r_i - c_k\} \quad \text{(C1b)}$$

Processors for which $O_j>0$ will be referred to as *overflow* processors, while *underflow* processors are those for which $U_k<0$. In an optimal solution we have

$$\sum_{i \in P_j} r_i^s = c_j, \text{ for all } j \text{ for which } O_j>0.$$

and the error *E* is equal to the sum of the total processor overflow

$$E = \sum_{i=1}^{N} \left| r_i - r_i^s \right| = \sum_{j \in \Gamma} O_j \quad \text{(C2)}$$

where *Γ* is the set of overflow processors (underflow processors do not contribute to the error *E*).

Therefore, the minimization problem of Eq. (9) can be rewritten as

$$\min E = \min \sum_{j \in \Gamma} O_j \qquad \text{(C3a)}$$

subject to

$$\sum_{i \in P_j} r_i^s \le c_j, \; P_j = \{i : T_i \; \text{scheduled on} \; j \; \text{processor}\} \qquad \text{(C3b)}$$

Eq. (C3) states that scheduling the tasks with rates as close as possible to their respective fair rates (Eq. (9)) is equivalent to finding a solution that minimizes the overall processor overflow. However, the minimization of Eq. (C3a) subject to the constraint of (C3b) is computationally intensive (it is the similar to the bin-packing problem, which is NP-complete), since every possible task assignment to the *M* available processors should be examined. In what follows, we propose a heuristic task rearrangement scheme of polynomial time, to obtain a good assignment of tasks to processors.

C.1 Processor Assignment

The proposed algorithm combines processors of capacity overflow with processors of capacity underflow to obtain a better exploitation of the overall processor capacity. More specifically, given an assignment of tasks to processors, we consider the rearrangement where a task of rate r_l assigned to an overflow processor is substituted for a task of rate r_m assigned to an underflow processor. After the task rearrangement, the overflow (underflow) capacity of the processors is updated as follows

$$R_j = O_j - \varepsilon \qquad \text{(C4a)}$$

$$R_k = U_k - \varepsilon \qquad \text{(C4b)}$$

where $\varepsilon = r_m - r_l$ (C4c)

Eq. (C4c) expresses the task rate difference between the two selected tasks, while R_j and R_k are the updated processor residuals. If $R_j > 0$, processor *j* remains at the overflow state after the task rearrangement, while if $R_j < 0$ processor *j* turns to the underflow state.

The tasks with rates r_l and r_m that are exchanged are selected so as to reduce the overall processor overflow, or equivalently reduce the error *E* given by Eq. (C2). A reduction is accomplished only if the task rate difference as expressed by Eq. (C4c) satisfies the following equation

$$\varepsilon : O'_j + O'_k < O_j \qquad \text{(C5)}$$

where $O'_j = \max(0, R_j)$ and similarly $O'_k = \max(0, R_k)$.

The explanation of Eq. (C5) is the following: Initially, processor *j* is overflow and processor *k* is underflow, so that only processor *j* contributes to the error *E* (see Eq. (C3)). After the task rearrangement, the total contribution of the processors *j* and *k* to the error *E* should be less that their initial contribution, for the rearrangement to yield an error reduction. Such rearrangements between overflow and underflow processors are repeated until no task rearrangement that satisfies Eq. (C5) can be found.

Appendix D Weighted Fair Queuing (WFQ)

Weighted Fair Queuing (WFQ) is a packet scheduling technique allowing guaranteed bandwidth services. The purpose of WFQ is to let several sessions share the same link. WFQ is an approximation of Generalized Processor Sharing (GPS) which, as the name suggest, is a generalization of Processor Sharing (PS). In PS each session has a separate FIFO queue. At any given time the *N* active sessions (the ones with non-empty queues) are serviced simultaneously, each at a rate of $1/N^{th}$ of the link speed. Contrary to PS, GPS allows different sessions to have different service shares. GPS have several nice properties. Since each session has its own queue, an ill-behaved session (who is sending a lot of data) will only punish itself and not other sessions. Furthermore, GPS allows sessions to have different guaranteed bandwidths allocated to them. In [16] Parekh and Gallager showed that when using a network with GPS switches and a session that is leaky bucket constrained an end-to-end delay bound can be guaranteed.

GPS is an idealized fluid model not practically realizable. WFQ, first presented in [17], is a packet approximation of GPS. In WFQ a packet at a time is selected and outputted among the active sessions. Each arriving packet is given virtual start and finish times. The virtual start time *S(k,i)* and the virtual finish time *F(k,i)* of the k^{th} packet in session *i* are computed as follows:

$$S(k,i) = \max(F(k-1,i), V(a(k,i)))$$

and

$$F(k,i) = S(k,i) + \frac{L(k,i)}{r(i)},$$

with *F(0, i)* = 0. *a(k, i)* and *L(k, i)* are the arrival time and the length of the packet respectively. *V(t)* is the virtual time function representing the progression of virtual time in the simulated GPS model and is defined as this:

$$\frac{dV(t)}{dt} = \frac{1}{(\text{Sum of active sessions shares at time t})}.$$

This means that when there are inactive sessions the virtual time progresses faster. In the corresponding GPS case, this can be viewed as that the remaining active sessions get more service. The packet selected for output is the packet with the smallest virtual finish time.

Appendix E Parekh-Gallager Theorem

The Parekh-Gallager showed that the use of Generalized processor Sharing (GPS), when combined with Leaky Bucket admission control, allows the network to make a wide range of worst-case performance guarantees on throughput and delay. Also because GPS isn't really possible to implement, they presented a practical implementation of the GPS called Packet-by-Packet Generalized Processor Sharing (PGPS), first proposed by Shenker, and Keshav under the name Weighted Fair Queueing (WFQ). The Parekh-Gallager theorem is stated below:

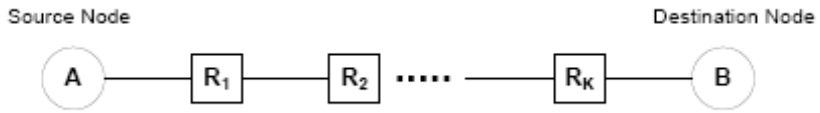


Figure 62: The Data Network in the Parekh-Gallager theorem

Consider a network path between a source node *A* and a destination node *B* (Figure 62), consisting of *K* intermediate routers. Every router along the path uses a Weight Fair Queueing (WFQ) scheduler and assigns different weights in a connection from node *A* to node *B*. Lets *g* is the least bandwidth allocated from the routers to the connection. Also we assume that the last *K*th scheduler has output link rate equal to *r(k)*, so:

$$g \le r(k)$$

Furthermore the source node *A* is leaky bucket constrained *(ρ, σ)*, where *ρ* is the maximum packet generation rate of the user and *σ* the maximum size of a packet burst, that the user can send. So that in time $[t_1, t_2]$ no more than $\rho \cdot (t_2 - t_1) + \sigma$ length of data are sent. We assume that:

$$g \ge \rho$$

Also let the size of the largest packet allowed in the network is *P* and the largest size of packet allowed on this connection is *M* bytes. Then the end to end delay that a packet experiences from *A* to *B* equals to:

$$end\_to\_end\_delay \le \frac{\sigma}{g} + \sum_{k=1}^{K-1}\frac{M}{g} + \sum_{k=1}^{K}\frac{P}{r(k)}$$

The first term $\frac{\sigma}{g}$ is the delay experienced by the last packet of a maximum burst of length *σ*, arriving at the WFQ scheduler of the first router. The subsequent routers receive no burst, so this term is accounted for only once. The second term $\sum_{k=1}^{K-1}\frac{M}{g}$ is the total delay experienced by the packet of size *M* for going though the next *K-1* routers, assuming that the routers use Generalized Processor Sharing (GPS). Finally the third term, $\sum_{k=1}^{K}\frac{P}{r(k)}$ is the correction term for using WFQ instead of GPS in the routers. That is in every router a packet will be transmitted at most $\frac{P}{r(k)}$ seconds after its theoretical GPS time.

Appendix F SAMCRA meta-code

F.1 Meta-code

```
1     SAMCRA(S,D)
2     MAX_LENGTH ← 1, Q ← {□}
3     for all nodes v do
4           K[v] ← 0
5           Store Dijkstra look-ahead info for all constraints in b[v]
6     for all constraints do
7           if    l∞(Dijkstra path from S → D for current constraint) < MAX_LENGTH
8           then  MAX_LENGTH ← l∞(Dijkstra path from S → D for current constraint)
9     priority(S[1]) ← 0, path(S[1]) ← NULL
10    insert(Q,S[1])
11    while Q ≠{□} do
12          u[i] ← extract_min(Q)
13          u[i] marked GREY
14          if    u = D
15          then  stop and return path(u[i])
16          else  for all v (□ adjacency_list(u) do
17                      if      v (□ path(u[i])
18                      then    PATH ← path(u[i]) + (u → v)
19                              DOMINATED ← dominated(PATH)
20                              mark all obsoleted paths v[j] BLACK
21                              PRED_LENGTH ← l∞(d[PATH] + d[b[v]])
```

```
22                          if    PRED_LENGTH ≤ MAX_LENGTH and DOMINATED = false
23                          then  if    all v[j] ≠ BLACK
24                                then  K[v] ← K[v] + 1
25                                      path(v[K[v]]) ← PATH
26                                      priority(v[K[v]]) ← PRED_LENGTH
27                                      insert(Q,v[K[v]])
28                                else  path(v[BLACKk]) ← PATH
29                                      decrease key(Q,v[BLACK_k], PRED_LENGTH)
30                                      if    v = D and PRED_LENGTH < MAX_LENGTH
31                                      then  MAX_LENGTH ← PRED_LENGTH
```

F.2 Discussion

Lines 1–10 take care of the initialization. For each node v, Dijkstra look-ahead information b[v] is computed and K[v], the number of stored sub-paths between **S** and v, is initialized to 0. If the length of a Dijkstra path for one of the constraints is smaller than the maximum length MAX_LENGTH, the value of MAX_LENGTH is updated. Furthermore, the source node **S** is inserted in the priority queue Q with an empty path history. The iterative search for the shortest path starts at line 11. First, the sub-path with the lowest predicted end-to-end path length is dequeued and marked ”GREY”, which means this path is not considered anymore during the next iterations. If the destination has been reached, the algorithm stops. Starting at line 16, the current sub-path is extended by examining all nodes that are adjacent to the current intermediate router and are not listed in the sub-path history. For each of these path extensions, path dominance is evaluated and previously computed sub-paths that have become obsolete, are marked ”BLACK”. If the extended path is non-dominated and the predicted length is less than or equal to the maximum length, the priority queue is updated. When an arbitrary sub-path v[$BLACK_k$] has become obsolete, it is replaced by the new sub-path in the priority queue (by decreasing the key value and updating the path). Otherwise, a new item is created and inserted into the priority queue. On lines 30–31, the maximum length MAX_LENGTH is updated if a shorter path reaching **D** is found.