European Research Network on Foundations, Software Infrastructures and Applications for large scale distributed, GRID and Peer-to-Peer Technologies
A Network of Excellence funded by the European Commission

Modeling and Supporting Grid Scheduling

A. Pugliese
apugliese@deis.unical.it
D. Talia
talia@deis.unical.it
DEIS
Universit della Calabria
87036 Rende, Italy

R. Yahyapour
ramin.yahyapour@unido.de
Institute for Robotics Research
University of Dortmund
44221 Dortmund, Germany

CoreGRID Technical Report
Number TR-0056
August 22, 2006

Institute on Resource Management and Scheduling
Institute on Knowledge and Data Management

CoreGRID - Network of Excellence
URL: http://www.coregrid.net

CoreGRID is a Network of Excellence funded by the European Commission under the Sixth Framework Programme

Project no. FP6-004265

Modeling and Supporting Grid Scheduling

A. Pugliese
apugliese@deis.unical.it
D. Talia
talia@deis.unical.it
DEIS
Universit della Calabria
87036 Rende, Italy

R. Yahyapour
ramin.yahyapour@unido.de
Institute for Robotics Research
University of Dortmund
44221 Dortmund, Germany

CoreGRID TR-0056
August 22, 2006

Abstract

Grid resource management and scheduling are important components for building Grids. They are responsible for the selection and allocation of Grid resources to current and future applications. Thus, they are the building blocks for making Grids available to user communities. In this paper, we briefly analyze the requirements of Grid resource management and provide a classification of schedulers. Moreover, we provide a formal model for the Grid scheduling problem and discuss the corresponding aspects for architectural implementations.

1 Introduction

Grids are geographically distributed computational platforms composed of heterogeneous machines that users can access via a single interface, providing common resource-access technology and operational services across widely distributed and dynamic *virtual organizations*, i.e., institutions or individuals that share *resources* [12, 18]. Resources are generally meant as reusable entities employable to execute applications, and comprise processing units, secondary storage, network links, software, data, special-purpose devices, etc. Grid computing differs from conventional distributed computing as it focuses on large-scale coordinated resource sharing by independent providers in different administrative domains. Grids are conceived to support innovative applications in many fields, and they are today mainly used as effective infrastructures for distributed high-performance computing and data processing. However, their application areas are shifting from scientific computing towards industry and business-related applications.

Grid resource management and scheduling are important components for building Grids. They are responsible for the selection and allocation of Grid resources to current and future applications. Thus, they are the building blocks for making Grids available to user communities. In the following, we will briefly analyze the requirements of Grid resource management and provide a classification of schedulers. Next, we will provide a formal model for the Grid scheduling problem and discuss the corresponding aspects for architectural implementations.

This research work is carried out under the FP6 Network of Excellence CoreGRID funded by the European Commission (Contract IST-2002-004265).

1.1 Grid resource management

The term *Grid Resource Management* (*GRM*) [5, 21, 25] specifies the set of functionalities that are needed for including the available pools of resources in a Grid environment and for providing means to access these resources. As Grid resources often happen to be very heterogeneous with respect to their owners/providers, access policies, performances, and costs, Grid resource management systems must take into account such heterogeneity (and possibly leverage it) to make resources efficiently exploitable, also providing support for performance optimizations.

Some of the main requirements of GRM systems are the adaptability to different application domains, scalability and reliability, fault tolerance, and adoption of dynamic information monitoring and storing schemes [25, 14]. The Global Grid Forum [12] is currently working on the definition of a general service-based resource management architecture for Grid environments, providing services for, e.g.: maintaining static and dynamic information about data and software components, network resources, and corresponding reservations; monitoring jobs during their execution; associating resource reservations with costs; assigning tasks to resources. Each of these services may have a different internal organization (centralized, hierarchical, peer-to-peer, etc.). The Open Grid Service Architecture (OGSA) as defined in the Global Grid Forum provides a general model towards a service platform for Grids [25, 14] which includes resource management functionalities.

For this paper, we use the following terminology for Grid resource management systems. *Tasks* are often referred to as the basic building blocks of Grid applications (usually called *jobs*). Jobs are composed of (possibly dependent) tasks, which are coordinated by GRM systems for their execution. This includes search and selection of suitable and available resources as well as the temporal coordination and facilitation of the execution itself. A task, when executed on a certain host, may generate a number of local processes (Figure 1) on which such high-level GRM systems may generally have limited (if any) control. The same control limitation exists over other resource types, such as locally-managed networks, secondary storage devices, etc.

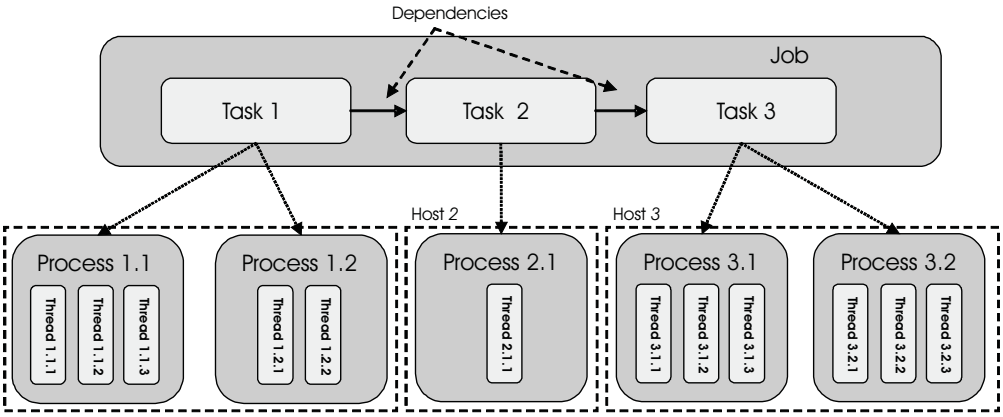

Figure 1: Jobs, tasks, and local processes

Due to the dynamic nature of Grids and the lack of information on resources and jobs, the reliability of job execution is a crucial topic for GRM systems. Here, often advance information on resource availability and performance are required. *Service Level Agreements* (*SLA*s) are typically considered as suitable means to support the necessary reliability. Thus, resource reservations and Quality-of-Service guarantees are being pointed out as a major requirement of modern GRM systems [13]; however, even in the absence of SLAs, some other approaches have been proposed to improve reliability and performance of resources including e.g. performance prediction, job replication, dynamic adaption to performance changes [3, 29, 34].

One of the most important features a GRM system is the capability to "suitably" assign tasks to resources; in fact, in realistic Grid applications, it may generally be infeasible for users to manually find and specify all the needed resources during the composition of jobs. The dynamic nature of Grids, in which resource availability can change constantly, requires support of an automatic way to assign tasks to resources. This feature is typically provided by *schedulers*, i.e., sets of software components or services which, on the basis of knowledge or prediction about computational and I/O costs, generate *schedules*, i.e., assignments of tasks to resources along with their timing constraints, with the objective of optimizing a certain performance function [27]. The timing constraints are not only based on the task itself but include the dynamic resource situation as well; typically, computing resources are operated in a space-sharing fashion

with a queuing system controlling the access. It is the task of a scheduling system to consider and synchronize the task execution with its local resource situation.

On the basis of the performance function adopted, schedulers can be classified into different categories:

- *task-oriented* schedulers (also known as *high-throughput* schedulers) adopt performance functions related to the overall system performance in executing the workload.
- *resource-oriented* schedulers aim at optimizing resource utilization or at imposing fairness criteria; here, the focus relies on the resource-oriented performance criteria;
- *job-oriented* (also known as *application-oriented*) schedulers concentrate on the improvement of the performance of individual jobs.

Moreover, schedulers may be classified on the basis of the level at which they work:

- *process-level* schedulers are OS-level components that coordinate the execution of local processes and deal, e.g., with efficient local CPU usage (such schedulers are generally not considered part of GRM systems);
- *task-level* schedulers coordinate the execution of single tasks;
- *resource-level* schedulers deal with resource usage;
- *meta-schedulers* coordinate among other, usually local, schedulers, i.e., they examine a set of potential schedules for jobs to be run on different resources and determine an overall schedule.

In Grid environments, all of these kinds of schedulers must co-exists, and they could in general pursue conflicting goals. For instance, improving the performances of a single job can worsen those of other jobs and of the overall system (and vice versa). Similarly schedulers operating at different levels might fail to adequately take into account the activities of another scheduler. That is, planning for future task allocations on resources can fail as resources are also considered by other schedulers. Thus, there is need for implicit or explicit interaction between the different schedulers in order to execute the tasks.

1.2 Scheduling models

To properly describe scheduling scenarios, a suitable *scheduling model* must be adopted. In general, a scheduling model comprises (Figure 2):

- A *resource model*, describing characteristics, performance, availability and cost of resources. Important requirements of this model are the timeframe-specificity of predictions, dynamicity, and capability to adapt to changes. This also includes the consideration of different level of dynamicity in the resource information.
- A *job model*, provided to users for describing jobs. This model may be based on dataflow or workflow formalisms, that mainly resort to task dependency graphs, or may use formalisms that are specifically targeted to particular domains [23].
- A *performance metric*, for measuring the performances to be improved.
- A *scheduling policy*, comprising (*i*) a *scheduling process*, i.e., the sequence of actions to be taken on relevant events (e.g. new job submission or resource availability), and (*ii*) a *scheduling algorithm*, defining the way in which tasks are assigned to resources. The main requirement of the scheduling policy is the practical applicability. That is, a good tradeoff between effectiveness and efficiency, as the overhead of scheduling activity should be in reasonable balance with the execution of the task itself. It is well known that the scheduling in Grids is very complex, see also the later discussion. Therefore the search of a mere optimal task assignment can be computational hard or might often be infeasible. Thus, a sensible tradeoff is required to provide suitable scheduling solutions in an online operation in a fair amount of time in relation to the actual task execution.
- A *programming model*, provided to external components/environments for interacting with the scheduler and describing detailed features of an application programming.

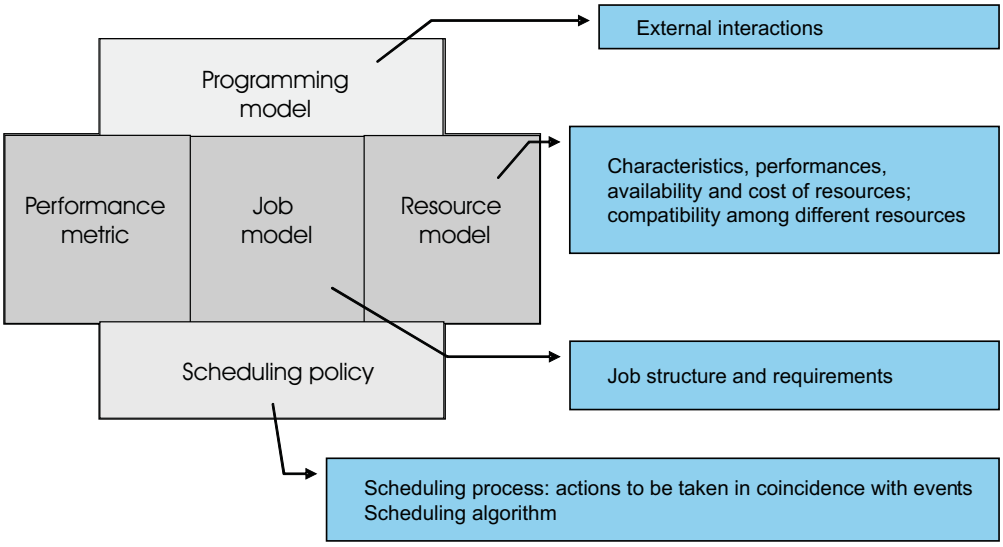

Figure 2: Structure of a scheduling model

1.3 Contributions

This paper is intended to support the work on common Grid scheduling models, their implementation and interoperability issues. Current research work shows different approaches on Grid scheduling caused by different Grid scenarios and requirements. Also for the future it is not foreseeable and expectable that certain scheduling solutions will dominate in the Grid world. Moreover, different application requirements and different Grid networks will require specific scheduling solutions. However, the development of Grids can highly benefit from the availability of common models and the existence of specific scheduling solutions that can inter-operate within these models. The presented approach is intended to support and advance this discussion in the Grid community.

To this end, we propose in this paper a general and extensible scheduling model, aimed at capturing the heterogeneity of Grid resources and applications, and the large number of variables that influence jobs' performances. The model is simple yet powerful, in that resource descriptions are enriched with extensible metadata allowing to embed the specific information needed in different scheduling scenarios.

Moreover, we propose a suitable architecture for the support of the model and the interaction with schedulers based on it. This architecture has been implemented and is available as a Java 5.0 library. The extensibility of the architecture is "naturally" induced by the model itself, which imposes only few constraints on the particular kinds of jobs to be scheduled and on the specific scheduling functionalities. The architecture is also designed to be open to the connection to external services, possibly provided by other Grid resource management instances.

The remainder of the paper is organized as follows. Section 2 describes our general scheduling model and provides a characterization of the scheduling problem induced by the model. Section 3 describes the scheduling architecture we designed for supporting the model. Finally, Section 5 outlines current and future works.

2 A general scheduling model

In this section we describe our general scheduling model. The model is heavily based on the use of extensible metadata in the form of attribute/value pairs. The main idea is the association of metadata with the description of both resources and jobs, so that particular scheduling functionalities (which are not part of the model), can leverage and/or extend them to capture the information they are based on.

2.1 Resource model

The resource model provides the abstractions for describing characteristics and performances of hosts, datasets, software components and network links. In the model, metadata about resources are expressed through $\langle attribute, value \rangle$

pairs; we denote with $\mathcal{P}$ the set of all possible such pairs.

We call *abstract* resources those having a description that is specified partially w.r.t. the one expected by execution services, or only indicates minimal requirements. Partial resource descriptions provide the degrees of freedom the scheduler bases on when making its decisions.

A *host* is modeled as a pair

$$\langle\, hostID,\, description \,\rangle$$

where $description \subseteq \mathcal{P}$ is a set of attribute/value pairs describing, e.g., the host's location, processor, operating system, memory, etc.

A *dataset* is a triple

$$\langle\, datasetID,\, description,\, hostID \,\rangle$$

where $description \subseteq \mathcal{P}$ may provide information about the modalities for accessing the dataset (location, size etc.), its logical and/or physical structure, etc.; $hostID$ is the identifier of the host at which the dataset resides.

Finally, a *software component* is a triple

$$\langle\, softwareID,\, description,\, hostID \,\rangle$$

where $description \subseteq \mathcal{P}$ may include the kind of data sources the software works on, the structure of its outputs, etc.; $hostID$ identifies the host where the software component code resides.

Note that the replication of datasets and software components is not explicitly handled by the model. Thus, each specialization can therefore exploit the extensibility of the model in adopting its replica maintenance mechanism. For instance, a basic mechanism to track the existence of several copies of a same dataset would be that of associating a global unique key to the dataset and mention it in its description.

In the remainder, we shall refer to sets of datasets, software components and hosts, using letters $\mathcal{D}$, $\mathcal{S}$ and $\mathcal{H}$, respectively. Moreover, $\mathcal{D}_\perp = \mathcal{D} \cup \{any\}$ will denote a set of hosts augmented with a special symbol any, whose meaning is explained in the following; the same applies to $\mathcal{S}_\perp$ and $\mathcal{H}_\perp$.

Resource performances are modeled using three estimation functions having the following structure:

- $\epsilon : 2^{\mathcal{D}}_\perp \times \mathcal{S} \times \mathcal{P}_\perp \times \mathcal{H}_\perp \times \mathbb{N}_\perp \to \mathbb{N}$ returning computation times, i.e., associating a quintuple $\langle\, ds,\, s,\, p,\, h,\, t \,\rangle$ with the time needed to execute software s on host h with input datasets ds and parameters p, starting at time t.
- $\nu : \mathcal{H} \times \mathcal{H} \times \mathbb{N} \times \mathbb{N}_\perp \to \{\mathbb{N} \cup \infty\}$ returning communication times, i.e., associating a quadruple $\langle\, h_s,\, h_d,\, d_s,\, t \,\rangle$ with the time needed to transfer a dataset of size d_s from host h_s to host h_d, starting at time t.
- $\omega : 2^{\mathcal{D}}_\perp \times \mathcal{S} \times \mathcal{P}_\perp \to 2^{\mathbb{N}}$ returning output sizes, i.e., associating a triple $\langle\, ds,\, s,\, p \,\rangle$ with the sizes of the outputs produced by software s when executed, with parameters p, on input datasets ds.

The any symbol is added to some of the sets in order to capture those scenarios in which the estimated quantities do not depend on some of the functions' parameters. For instance function $\nu(h_s,\, h_d,\, d_s,\, any)$ would evaluate communication costs in a network whose performance is assumed to be time-independent. Moreover, the structure of functions ϵ and ν imposes a discretization of the time scale (that is also used in the timing function defined in the following). This does not limit the generality of the model, but imposes suitable adjustments of the time scale (i.e., choice of a proper starting time and period) to capture the real conditions of resources over time.

Finally, the model considers the constraints on the assignability of tasks to resources, implicitly contained in their descriptions, through a *resource compatibility function*

$$\gamma : 2^{\mathcal{D}}_\perp \times \mathcal{S}_\perp \times \mathcal{P}_\perp \times \mathcal{H}_\perp \to \{true,\, false\}$$

where $\gamma(ds,\, s,\, p,\, h) = true$ if and only if software s can execute on host h with input datasets ds and parameters p.

Running examples

As running examples, we consider two different scheduling scenarios. In the first one, the Grid is tree-structured and the computational load is *divisible*, that is, it can be freely split among different machines (this happens, for instance, when subsets of an input dataset can be extracted and elaborated by different machines running the same software). Moreover, each machine has a different computing capability and a different network link to each of its children [28]. In this scenario, the overall computing load starts from the root machine, that keeps only a fraction of it and assigns the rest to its children. Each child recursively splits its load among itself and its children; the outputs produced are left onto the corresponding machines.

Here, for each host h, and for each child h_i, if T_h is the total load assigned to h, and f_iT_h is the fraction of T_h assigned to h_i, the time to process f_iT_h is $f_iT_h/p_i + f_iT_h/z_i$, where p_i is the speed of h_i and z_i is the network speed from h to h_i (note that p_i must indicate the *equivalent* computing power of the overall subtree rooted at h_i). The objective of the scheduling activity is that of minimizing the overall processing time; this is typically done by each node, at the time of arrival of its assigned load.

In this case, the proposed resource model can be instantiated as follows. At each node h of the tree, the set of known hosts comprises h and all of its children h_i, the input dataset ds^{T_h} corresponds to a load of T_h, and on each host there is a copy of the used software. We denote with s_{dl} the identifier of one copy of such software; the associated descriptions will recall that all the softwares are equivalent.

Since outputs are not moved from their locations, function ω is not used in this scenario. Moreover, no resource incompatibilities are assumed, therefore function γ always evaluates to $true$. The time to process f_iT_h is given by the formula $\epsilon(ds^{f_iT_h}, s_{dl}, any, h_i, any)+\nu(h, h_i, |ds^{f_iT_h}|, any)$, where dataset $ds^{f_iT_h}$ is the fraction of ds^{T_h} corresponding to a load of f_iT_h.

The second scenario that we consider is also quite common and has been addressed in many works regarding scheduling in heterogeneous environments [22]. In this scenario, the network can have an arbitrary topology, and the hosts have different computing capabilities (but each of them can execute each task). Moreover, the tasks may have precedence dependencies, but each task has a fixed *computation cost*, and its output sizes are known in advance (yielding *communication costs*). The objective of the scheduling activity is again the minimizing of the overall processing time.

In this example, the resource model instantiates as follows. If a task t_i runs, on host h_i, a software s_i with input dataset d_i, and its output must be moved to host h_j, the time to process t_i is given by $\epsilon(d_i, s_i, any, h_i, any)$, whereas the time to transfer its output is $\nu(h_i, h_j, \omega(d_i, s_i, any), any)$. Obviously, computation costs are used for evaluating function ϵ, whereas communication costs can be directly used as function ω. Again, no resource incompatibilities are assumed, so function γ always evaluates to $true$.
We will continue this running example in the following sections.

2.2 Job model

The job model views *tasks* as a tuples of the form

$$\langle taskID,\ datasetIDs_{in},\ datasetIDs_{out},\ softwareID,\\ parameters,\ description,\ hostID\rangle$$

where $datasetIDs_{in}$ (resp., $datasetIDs_{out}$) is a set of input (output) dataset identifiers, $softwareID$ identifies the software component to be executed, $parameters \subseteq \mathcal{P}$ defines the parameters of the particular execution, and $description \subseteq \mathcal{P}$ contains metadata about the task. $hostID$ identifies the host at which the task is to be executed. In the case of software movement tasks, a $datasetID$ must also be associated with the software moved. A task is said to be abstract if it is partially specified, or if it refers to abstract resources.

Finally, a *job* is a pair

$$\langle(\mathcal{T},\ \mathcal{A}), description\rangle$$

where: (i) $(\mathcal{T},\ \mathcal{A})$ is a *task precedence* directed acyclic graph, with $\mathcal{T}$ being the set of tasks, and $\mathcal{A} \subseteq \mathcal{T} \times \mathcal{T}$ the set of arcs, each arc $(t_i,\ t_k) \in \mathcal{A}$ dictating that task t_k depends on task t_i, i.e., it uses one of its outputs, so its start can not be

scheduled before the completion of t_i (and after other possibly needed data movement tasks); (ii) $description \subseteq \mathcal{P}$ contains metadata about the job.

A job is said to be abstract if it contains at least one abstract task. Abstract tasks need to be instantiated before being really executed, so they are the target of the scheduling activity. The job may also contain *implicit* data movement tasks, i.e., it is allowed to define computational tasks by only indicating their components, even if this implicitly requires the movement of softwares and/or datasets from/to different hosts. Obviously, the needed movement tasks must be generated and added to the job before ordering its execution. Therefore, schedules must always be completed through *decoding*, i.e., through an analysis of the dependencies among computational tasks, to find implicit data movement tasks to be added.

In the remainder, we adopt dot notations, denoting e.g. with $s.h$ the host offering software s, with $t.d_k.h$ the host at which the k-th input dataset of task t resides, and so on.

2.3 Scheduling policies and performance metrics

The proposed scheduling model does not impose any constraints on the scheduling processes and algorithms employable, nor does it require the use of a particular performance metric. That is, different scheduling policies can be implemented with individual optimization goals. In computational Grids such goals are very often the minimization of turn-around time, maximization of throughput or cost consideration. However, Grid scenarios could also be conceived with arbitrary complex optimization goals, e.g. a multi-criteria optimization with individual utility functions for users and providers.

Therefore, in different scheduling scenario specific scheduling functionalities can be adopted, based on properly designed resource metadata. The model is designed to be open to different external interaction modalities, so it also remains abstract with regard to its programming models. The Java library associated to the supporting architecture proposed later in Section 3 represents one possible way of interacting with schedulers based on the proposed model.

It should be noted that, as a consequence of the openness to different scheduling processes, the model also does not make assumptions on their dynamicity. Therefore, *dynamic scheduling processes with re-scheduling* can be employed in which the scheduler is invoked initially and then, during the execution of jobs, it may be invoked repeatedly again as a consequence of the occurrence of new events. In this way it can re-schedule running tasks (if mechanisms are supported) as well as not yet executed tasks. Therefore, scheduling algorithms are allowed to produce *partial* outputs, i.e., schedules comprising unassigned tasks (called *pending*), to be scheduled subsequently; in this case, the associated processes must be prepared to receive information about the events occurred, to make the appropriate decisions.

2.4 General problem characterization

The proposed scheduling model, if adopted in its most general forms, quickly leads to instances of intractable scheduling problems; this is the drawback of its generality. However, in many real-world scenarios that are well captured by the model, there are scheduling problems for which efficient algorithms exist that produce reasonable results, e.g. within different guaranteed degrees of sub-optimality. Moreover, the model is capable of supporting the description of complex scheduling scenarios, typical in Grid environments, for which new techniques can be developed.
In the most general case, the input of the problem consists of

- three sets $\mathcal{D}^i$, $\mathcal{S}^i$, and $\mathcal{H}^i$, containing descriptions of datasets, software components, and hosts;
- functions ϵ, ω, ν, and γ (as defined in Section 2.1);
- a job $\langle(\mathcal{T}, \mathcal{A}), description\rangle$, where $\mathcal{T} = \mathcal{T}_c \cup \mathcal{T}_m$, with $\mathcal{T}_c$ denoting the set of computational tasks and $\mathcal{T}_m$ the set of data movement tasks;

whereas the output consists of

- three sets $\mathcal{D}^o$, $\mathcal{S}^o$, and $\mathcal{T}^o$ of datasets, software components, and tasks;
- a *timing function* $\sigma_T : \mathcal{T}^o \rightarrow \mathbb{N}$ associating each task with the time at which it must be started.

Sets $\mathcal{S}^o$, $\mathcal{D}^o$, and $\mathcal{T}^o$ contain instantiated elements for the subset of $\mathcal{T}^i$ that has been examined by the algorithm. In addition, $\mathcal{S}^o$ and $\mathcal{D}^o$ comprise resources generated or discovered during job execution, such as, e.g., generated output datasets, software moved or new available hosts; these resources are called *virtual*.

Obviously, $\mathcal{T}^o$ must be such that resource constraints are satisfied, i.e.,

$$\forall\, t^o \in \mathcal{T}^o,\, \gamma(t^o.ds,\, t^o.s,\, t^o.p,\, t^o.h) = true.$$

whereas function σ_T must satisfy precedence constraints, i.e.,

$$\forall (t_i,\, t_k) \in \mathcal{A},\, \sigma_T(t_i^o) + \tau_{t_i^o} + \tau_{[t_i^o,\, t_k^o]} < \sigma_T(t_k^o)$$

where $\tau_{t_i^o}$ is the duration of task t_i^o, and $\tau_{[t_i^o,\, t_k^o]}$ is the duration of a data movement task between t_i^o and t_k^o, if needed.

To deal with the potentially enormous solution space (in principle, exponential with the number of abstract tasks in $\mathcal{T}^i$) classical schedulers for distributed environments make strong assumptions about the underlying system. Typically, they assume to be in control of an entire completely-connected invariant resource pool, composed of processing units and network links having very similar (or even identical) performances (see e.g. [22]). Even under these assumptions, however, the resulting problem (called *precedence-constrained scheduling*) is $\mathcal{NP}$-hard if $\mathcal{H}^i > 1$ [11].

Moreover, as many of the above assumptions are not realistic in the inherently more heterogeneous Grid environments, our problem is significantly more general than classical precedence-constrained scheduling. Therefore, exact optimal techniques, such as integer-linear or constraint programming, are rarely usable as they incur in an exponential duration of the scheduling process; in this setting, the most suitable approaches must tackle the problem heuristically. Here, approaches exists to use market-oriented models or game theory as well as evolutionary algorithms to produce suitable schedules in limited time.

Running examples

In the divisible load scheduling scenario, in order to minimize the overall processing time, each child in the network tree splits its load among itself and its children. As pointed out in [28], each node in the tree can ideally be replaced by its equivalent, that is one whose computational power corresponds to the whole subtree rooted in that node. Therefore, optimal scheduling can be approached by making local decisions, i.e., each node looking at the equivalents of its children and optimizing a step of the computation.

Therefore, at each node h in the tree, the input job comprises of an independent task for each child h_i, plus one for h itself. Each task is assigned to the corresponding host, and the software identifier is that of the one residing on that host. The fraction of the input dataset to be assigned to each task must be decided; this corresponds to generating a virtual dataset on each host containing the proper portion of the input dataset, and adding a data movement task to the job for each h_i.

An example for an elegant and efficient (polynomial) scheduling algorithm for this problem is proposed by Robertazzi [28]. The algorithm is based on the intuition that the best schedule in this scenario is obtained when all the involved machines finish their assigned computations at the same time: a system of linear equations can be derived that minimizes the differences among processing times.

Our second example scenario deals with more complex job structures. Figure 3(top) shows a sample job description for this scenario, where node weights represent computing costs, and arc weights represent communication costs (see Section 2.1). A representation of the same job using the proposed job model is shown in Figure 3(bottom), where $taskID$s are represented as node labels.

Note that, using our job model, metadata about tasks comprise their computation costs, whereas communication costs are directly embedded as job metadata (not shown in the figure). Note also that the described job includes an implicit data movement task for each arc.

A very large number of heuristics aimed at finding good solutions to the resulting problem has been proposed, each having a different cost/benefit tradeoff. For a thorough survey of these heuristics see, e.g., [22].

3 Supporting architecture

The architecture of the scheduler comprises three main components, a *Mapper*, a *Cost/Size Estimator* (*CSE*), and a *Controller* (Figure 4). The main components of the architecture provide an open interface that allows plugging in internal modules that, adhering to specific interfaces, implement different functionalities.

The architecture components interact between one another, in a P2P fashion if residing at different locations. Moreover, they interact with resource information services, for gathering information about available resources, with

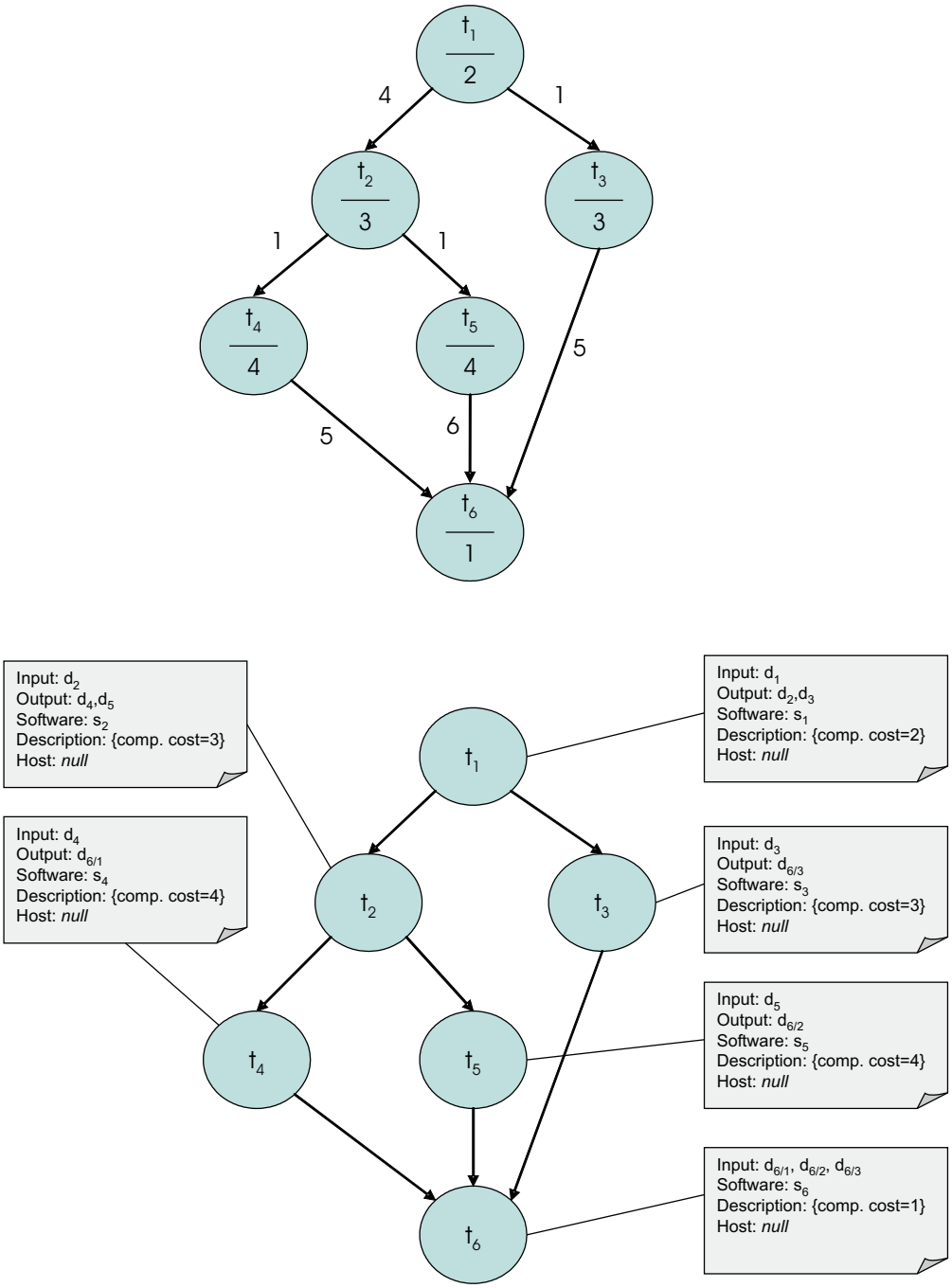

Figure 3: Example job descriptions.

execution management services, for submitting task execution requests, and with job supervision services, for monitoring job executions. Figure 5 shows an example of interaction scenario among scheduling instances and external services, all residing at different locations. In this example, the scheduling process in different Controllers interact with eachother to find suitable job allocations. This approach can be used to model Grid markets in a P2P fashion in which different scheduling process negotiate with eachother through request and offer protocols.

Mapper. The mapper computes schedules from (abstract) jobs. It embeds a *Scheduling Algorithm*, and a *Matchmaker* that implements the logic for matching partial resource descriptions with concrete ones. To make its decisions,

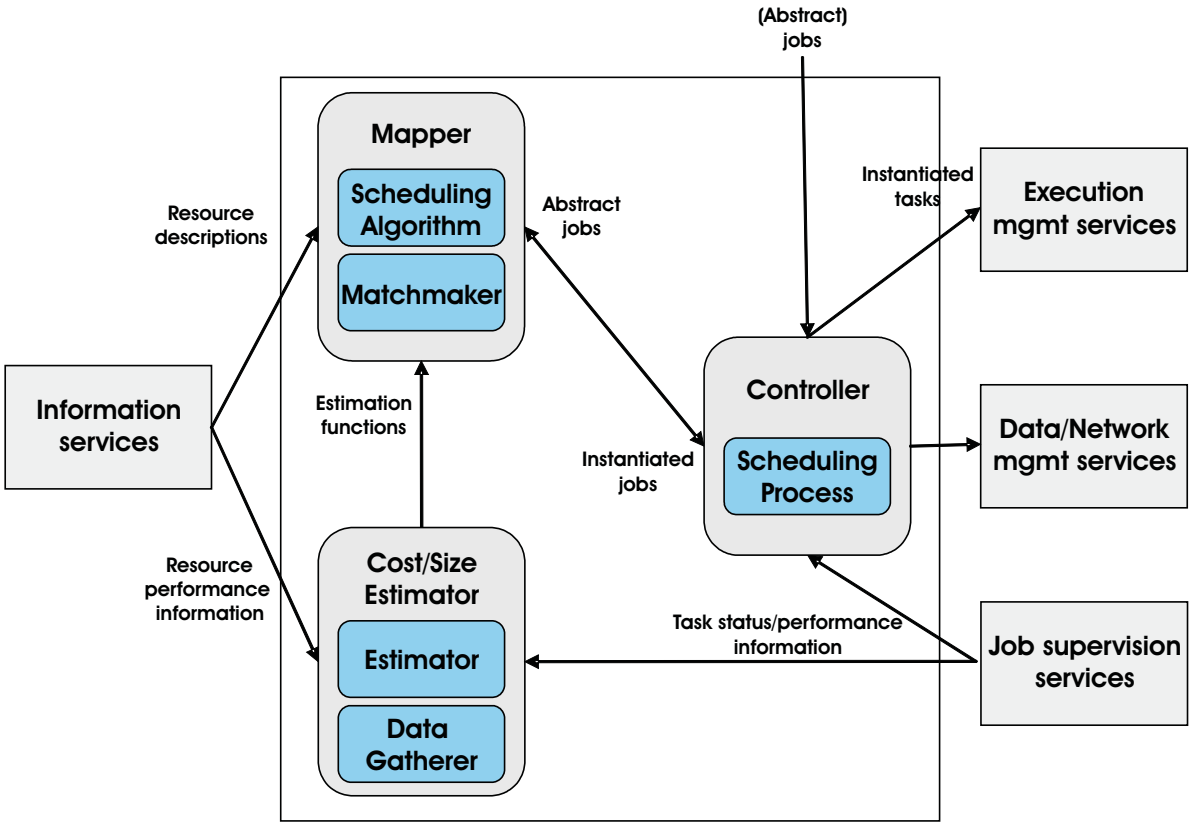

Figure 4: Scheduler architecture

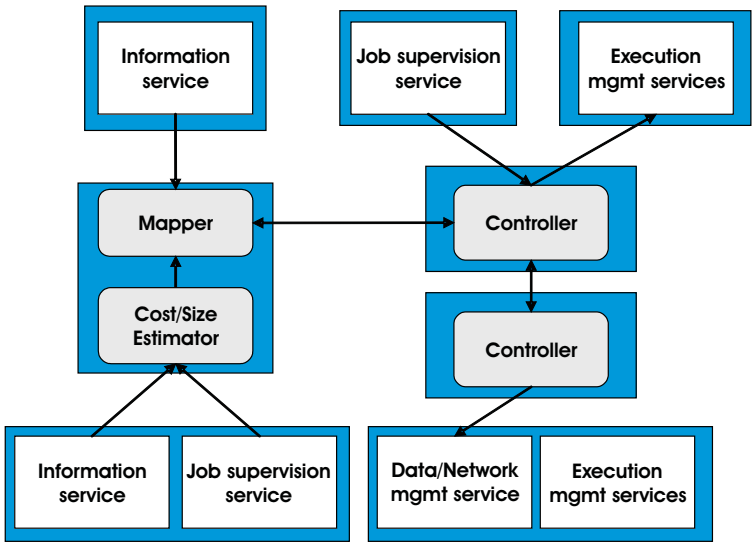

Figure 5: Example of interaction scenario (dark boxes correspond to hosts)

the mapper makes use of resource descriptions and computational and I/O cost evaluations.

Cost/Size Estimator. The cost/size estimator computes functions ϵ, ν, and ω, and makes them available to the mapper. The CSE comprises the *Data Gatherer* module, that feeds the resource information cache with collected information about current and future availability, performance and cost of resources. The *Estimator* module

deals with the actual calculation of the estimation functions on the basis of the perceived status of resources w.r.t. time.

Controller. The controller guides the scheduling activity. It receives (abstract) jobs, requests the corresponding schedules to the mapper, and orders the execution of scheduled tasks to the execution management services. During the execution of scheduled tasks, the controller also receives notifications from the job supervision services about significant events occurred, and re-schedules unexecuted parts of the job.

Resource information is maintained in a *Resource Information Cache* (not shown in Figure 4), that is heavily indexed to provide efficient access to dynamically-updated resource information. Synchronous and asynchronous information retrieval methods can be applied that suit the underlying Grid infrastructure, e.g., if P2P Grid systems are used, results from information requests are received asynchronously over time.

For each job to be scheduled, the scheduler performs the following logical steps:

1. gathering of information about characteristics and performances of available resources;
2. selection of resources usable to execute tasks, based on their descriptions and estimated performance information;
3. evaluation of a certain set of possible schedules and choice of the one optimizing the performance metric;
4. request of task execution;
5. supervision of the execution and adaptation of the schedule to new information about tasks' status and available resources.

3.1 Implementation issues

The presented model and its general architecture as presented above have been designed in a generic way. Note, that we did not limit the scope of the model to a particular implementation. As discussed in the introduction, the goal of the work is to stipulate the discussion towards common architectural designs that allows the creation of individual scheduling solutions which are interoperable within common frameworks. Thus, the presented model has been designed to be mappable to different Grid architectural models.

As mentioned in Section 1.1 there are several approaches to service-based Grid middleware. The most prominent activity is the Open Grid Service Architecture (OGSA) as defined within the Global Grid Forum [14]. The architectural model presented above is not in opposition to OGSA but can easily be mapped to the OGSA service landscape. Within OGSA [15] the logic of job scheduling and mapping is split among several services. The key services for this activity are the *Execution Planning Service (EPS)* and the *Candidate Set Generator (CSG)*. The CSG is responsible for searching and pre-selecting resources that fit a given resource requirement; however, it does not schedule nor interact with any planning services. The EPS is responsible for the decision making on where and when a task is executed. That is, the scheduling is mainly part of the EPS and to some minor extent of the CSG as its method to provide candidates influences the EPS results. The model presented in this paper can be mapped to the OGSA approach as the components in Figure 4 could be placed in an EPS implementation. The links to external services can be supplemented by the OGSA counterparts; for instance to the OGSA basic execution services or OGSA information services. The creation of service level agreements (SLAs) can be facilitated by the use of the WS-Agreement specification of the GGF [16].

As an extension to the OGSA model, our approach contains a model for the scheduling process and already considers implementations with P2P techniques, as shown in Figures 5.

There are additional efforts towards a common scheduling architecture within the GGF *Grid Scheduling Architecture research group (GSA-RG)* [17] and the *CoreGRID Institute on Resource Management and Scheduling* [6]. The presented models in these groups conceive abstract *scheduling instances* which can interoperate in various fashions to model different Grid scenarios. Again, our model is complementary to the approach of these groups. Here, our general scheduling model and our scheduler architecture can be seen as an implementation of the aforementioned *scheduling instance* with the same links to external services for information, execution management, data management and job supervision.

A library implementing the proposed architecture is currently available as a set of Java 5.0 packages. The library provides suitable data structures for representing tasks, jobs, and all the resources targeted by the scheduling activity.

The internal resource information cache is implemented as a set of hash structures, that provide efficient access, can be dynamically updated, and ensure availability by periodically dumping themselves to disk or to external database management systems.

The library provides the needed extensibility by defining Java interfaces for scheduling processes and algorithms, matchmakers, estimators, and data gatherers. Particular scheduling functionalities can therefore be realized as classes implementing the methods defined in those interfaces.

The controller, mapper, and cost/size estimator are implemented as regular Java classes. Through suitable methods, they can receive instances of their internal components, embed them, and coordinate their usage when receiving requests for specific functionalities. Their interactions with other architectural components are realized by means of *Java Remote Method Invocation* calls, in order to be open to remote usage, but can also be accomplished through local method calls, for better efficiency.

External components, such as execution management services etc., can be connected through the use of intermediate objects implementing specific interfaces, whereas objects providing particular scheduling functionalities needed by internal components can be connected through generic connection methods defined in their Java interfaces. Finally, the connections from one component to another can be freely moved using simple connection methods, thus leaving the necessary flexibility in choosing which components interoperate during the scheduling activity.

4 Example Implementation of an Application-oriented Scheduler within the *Knowledge Grid*

In this section we show an actual implementation example of how our general model and architecture have been specialized to create a task-level job-oriented scheduler. The implementation has be done in the context of the KNOWLEDGE GRID (*K-Grid*) [8]. The KNOWLEDGE GRID is an architecture built atop basic Grid middleware services that defines more specific knowledge discovery services. The services are organized in two hierarchic levels: the *Core K-Grid layer* and the *High-level K-Grid layer*, as depicted in Figure 6. In the following, the term KNOWLEDGE GRID *node* will denote a node implementing the KNOWLEDGE GRID services.

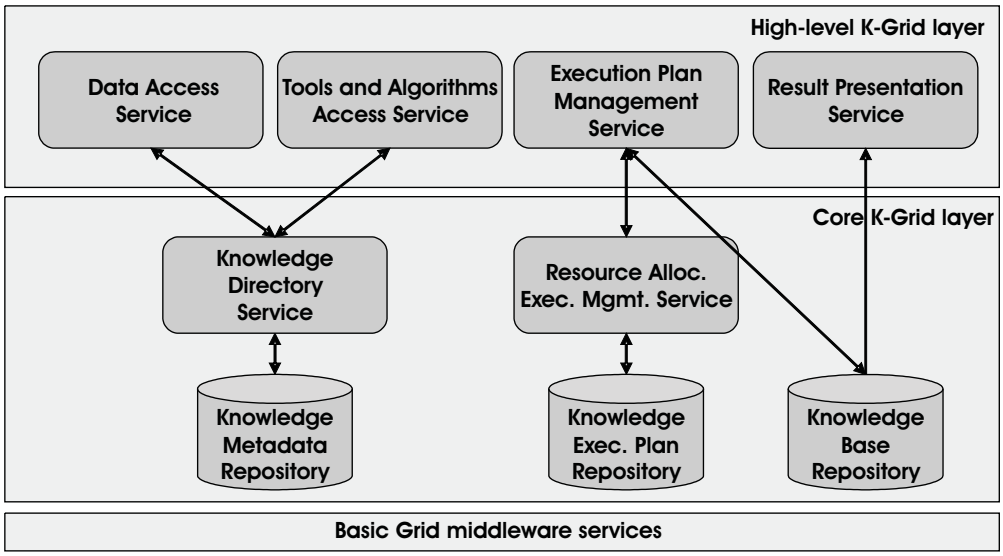

Figure 6: The KNOWLEDGE GRID architecture

The Core K-Grid layer offers the basic services for the definition, composition and execution of a distributed knowledge discovery computation over the Grid. The Core K-Grid layer comprises two main services: the *Knowledge Directory Service* (*KDS*) and the *Resource Allocation and Execution Management Service* (*RAEMS*). The KDS is responsible for maintaining metadata describing KNOWLEDGE GRID resources. The metadata information, represented in *XML*, is stored in a *Knowledge Metadata Repository* (*KMR*).

The RAEMS comprises three main modules: a *Scheduler*, an *Execution Manager* and a *Task Monitor*. The scheduler is used to find a suitable mapping between an abstract *execution plan* (job) and available resources, with the goals of (*i*) satisfying the constraints (computing power, storage, memory, network performance) imposed by the execution plan, and (*ii*) minimizing its completion time. The instantiated execution plan is then managed by the Execution Manager, that translates it into requests to basic Grid services, submits these requests and, after their execution, stores knowledge results in the *Knowledge Base Repository* (*KBR*). Finally, the Task Monitor follows the execution of submitted tasks, and notifies the scheduler about significant events occurred.

The High-level K-Grid layer includes services used to compose, validate, and execute a parallel and distributed knowledge discovery computation. Moreover, this layer offers services to store and analyze the discovered knowledge. The *Data Access Service* (*DAS*) is responsible for the search, selection, extraction, transformation and delivery of data to be mined. The *Tools and Algorithms Access Service* (*TAAS*) is responsible for searching, selecting and downloading data mining tools and algorithms. The *Execution Plan Management Service* (*EPMS*) manages execution plan represented by graphs describing interactions and data flows between data sources, extraction tools, DM tools and visualization tools. The EPMS allows to define the structure of an job by building the corresponding graph and adding a set of constraints about resources. Generated execution plans are stored, through the RAEMS, in the *Knowledge Execution Plan Repository* (*KEPR*). The *Results Presentation Service* (*RPS*) offers facilities for presenting and visualizing the knowledge models extracted (e.g., association rules, clustering models, classifications), and offers an API to store them in different formats in the Knowledge Base Repository.

As seen above, the KNOWLEDGE GRID scheduler is part of the Resource Allocation and Execution Management Service. Thus, each KNOWLEDGE GRID node has its own scheduler which is responsible for responding to scheduling requests coming from the same or other nodes (Figure 7). Obviously, different scheduler instances can also communicate with each other to exchange the information they need.

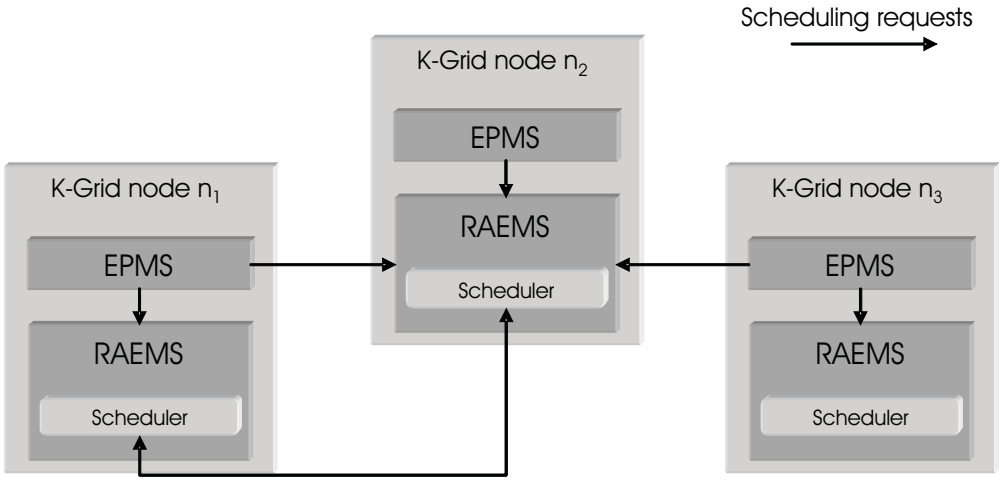

Figure 7: Scheduling-related interactions among different KNOWLEDGE GRID nodes

The scheduler currently provides the following built-in functionalities:

Scheduling algorithm. Several algorithms are provided (described in the next section), all potentially producing partial schedules.

Scheduling process. Events that fire the re-scheduling activity are: (*i*) the completion of all tasks preceding a pending task in the current schedule; (*ii*) important performance variations; (*iii*) task/host failures.

Data gathering. For the evaluation of current and future CPU availability, in the absence of service level agreements, the scheduler adopts as its information source the *Network Weather Service* (*NWS*) [32, 33], if present, or directly the hosts' descriptions. For evaluating the processing requirements of software components, the scheduler makes use of user-provided descriptions, if present, or a *sampling method* [26]. Finally, for assessing network performances (bandwidth and latency) the scheduler adopts the NWS as well, if present, or a simple probing technique [29, 33].

Cost estimation. For estimating computational costs, the scheduler employs the formula

$$\epsilon(ds,\, s,\, p,\, h,\, t) = \frac{req(ds,\, s,\, p)}{perf(h) \cdot avail(h,\, t)}$$

where $req(ds,\, s,\, p)$ represents the processing requirements (with respect to a reference host) of software s run with parameters p on datasets ds, $perf(h)$ is the no-load performance of host h, and $avail(h,\, t)$ is the fraction of processing cycles available on host h at time t (this process is based on task profiling and analytical benchmarking [19, 26]). Finally, the scheduler makes use of user-provided descriptions of the relationships between input and output sizes of software components.

4.1 Scheduling algorithms

Many interesting scheduling heuristics have been proposed in the past [1, 2, 4, 7, 10, 20, 22, 24, 30, 31, 35], both based on general-purpose ones as graph decompositions, genetic algorithms or "natural" heuristics (e.g. simulated annealing), and specific, e.g. using priority list and critical path concepts. The *NetSolve* system [2] employs a simple algorithm for scheduling independent tasks *online*, i.e. as soon as they are submitted to a system. Other approaches to scheduling independent tasks are presented in [1, 26]. The main limitation of such approaches is that they disregard dependencies among tasks; in fact, looking only at the tasks that are ready to be run at scheduling time may produce performance losses for subsequent tasks. In the context of the *GrADS* project [9] an interesting approach is presented that groups together hosts belonging to the same organizational domain (therefore close to one another in terms of "network distance") and schedules entire jobs to host groups. [30] discusses heuristics for scheduling task precedence DAGs in online and *batch* modes, i.e. looking at single tasks or entire subparts of the input DAG. Finally, in [24] natural heuristics are applied to workflow instances.

The mapper for the KNOWLEDGE GRID calculates schedules by first performing a pre-processing phase with the objective of reducing the size of the search space, then applying one or more of the provided heuristics to perform the actual mapping. The pre-processing phase comprises the following steps:

1. The input DAG is reduced by keeping only *entry* tasks (i.e. tasks that have all inputs ready at scheduling time) and tasks that depend on them up to a certain depth p.
2. The DAG is further reduced by keeping only the tasks whose input size is known or it can be evaluated using the ω function. The mapper makes this reduction essentially because we deal with data-intensive jobs, so the choice of hosts on which to schedule tasks cannot leave aside the associated communication costs.
3. In the spirit of [9, 24], by looking at the variations of ϵ w.r.t. its variable h, and of ν w.r.t. h_s and h_d (as these variations can be seen as indicators of machine and network heterogeneity, respectively), groups of similar hosts are formed through a simple clustering algorithm.
4. Finally, for each abstract task, the set of possible hosts on which that task can be scheduled is built, by looking at resource descriptions, in order to explore only feasible assignments.

The heuristics currently under evaluation are the following:

Min-Min. The Min-Min heuristic performs the following steps:

1. $schedulableTasks \leftarrow$ entry tasks in the DAG;
2. **repeat**
   (a) Evaluate the earliest completion times of each task in $schedulableTasks$ over all compatible hosts;
   (b) Select task $j_s \in schedulableTasks$ having the minimum earliest completion time and add it to the result along with the corresponding host;
   (c) Remove j_s from $schedulableTasks$;
   (d) Add to $schedulableTasks$ new tasks that become ready to be run after the completion of j_s;

   until $schedulableTasks = \emptyset$.

Max-Min. The Max-Min heuristic works in the same fashion as the Min-Min one, except that at step 2(b) it selects task j_s as the one incurring in the maximum earliest completion time.

Sufferage. The Sufferage heuristic works in the same fashion as the Min-Min one, except that at step 2(b) it selects task j_s as the one having the largest difference between best and second best completion times.

Minimum Completion Time (MCT). The Minimum Completion Time heuristic examines only one randomly-chosen task at a time:

1. $schedulableTasks \leftarrow$ entry tasks in the DAG;
2. **repeat**
   (a) Randomly select a task $j_s \in schedulableTasks$;
   (b) Evaluate the earliest completion time of j_s over all compatible hosts and add it to the result along with the corresponding host;
   (c) Remove j_s from $schedulableTasks$;
   (d) Add to $schedulableTasks$ new tasks that become ready to be run after the completion of j_s;

   until $schedulableTasks = \emptyset$.

Opportunistic Load Balancing (OLB). The OLB heuristic works in the same fashion as the MCT one, except that at step 2(b) it assigns j_s to the first host that becomes idle.

The presented Knowledge Grid scheduler is an example on how to utilize and specialize the general model in this paper. Similarly, different scheduling scenarios can be implemented. We are currently considering different approaches to cost estimation and different scheduling algorithms, in order to assess their performances with respect to classical measures, such as distance from the optimal solution and *turnaround* time (i.e., total execution time, comprising the scheduling activity itself), but also in terms of other properties which are particularly desirable in Grid environments, such as robustness w.r.t. unpredictable changes and stability. We are also designing a Grid service-based interfacing of the Java library implementing the proposed architecture, based on the *Web Services Resource Framework* standard.

5 Conclusion and future works

Within this paper the problem of grid scheduling has been formalized in a general and abstract model. For this model, a generic supporting architecture has been defined. This paper is intended to support the ongoing work on common Grid scheduling models, their implementation and inter-operability issues. It is generally considered that the automatic management by Grid scheduling is a crucial feature for wider adoption of Grids. However, while there are currently several independent scheduling solutions designed, there is no common standard on the model and the architecture in which these schedulers exist. Different application requirements and different Grid networks will require specific scheduling solutions. However, for a broad adoption of Grids it will be necessary that common models and architectures are available that support the easy integration of different scheduling strategies. Moreover, the interoperability of different scheduler implementations will be a crucial aspect for future Grid systems.

The presented approach is intended to support and advance this discussion in the Grid community. The presented model is not exclusive but flexible enough to be mapped to different Grid scenarios. As outlined in the paper, different specializations can be implemented. We presented as a proof of concept the example of the Knowledge-Grid scheduler.

We plan to investigate several extensions of the proposed model and architecture, among which:

- The use of an ontological organization of resource descriptions; ontologies could provide richer semantics to describe resources, jobs, and matching rules.
- The use of checkpoints, that is the possibility to suspend tasks and re-schedule their execution without restart, for improving performances or in case of failures.

- The adoption of resource cost functions, in order to consider the problem of choosing a set of resource reservations to be made for minimizing the cost of a job execution, given a maximum completion time.
- The integration of market-oriented strategies like e.g. the negotiation and auctioning of resource access. The presented model already supports this approach, however, a more detailed definition of the interaction between different schedulers could help the work towards common interfaces.

References

[1] A. Abraham, R. Buyya, B. Nath. Nature's heuristics for scheduling jobs on computational Grids. *The 8th IEEE International Conference on Advanced Computing and Communications*, 2000.

[2] D.C. Arnold, H. Casanova, J. Dongarra. Innovations of the NetSolve Grid computing system. *Concurrency and computation: practice and experience*, 14, 2002.

[3] Z. Balaton, P. Kacsuk, N. Podhorszki, F. Vajda. Comparison of Representative Grid Monitoring Tools. Tech. rep., Hungarian Academy of Sciences, 2000.

[4] T.D. Braun et al. A Comparison Study of Static Mapping Heuristics for a Class of Meta-tasks on Heterogeneous Computing Systems. *Heterogeneous Computing Workshop*, 1999.

[5] R. Buyya, S. Chapin, D. Di Nucci. Architectural models for resource management on the Grid. *First IEEE/ACM International Workshop on Grid Computing*, 2000.

[6] CoreGRID Technical Reports. "A Proposal for a Generic Grid Scheduling Architecture", N. Tonellotto, R. Yahyapour, Ph. Wieder, CoreGRID Institute on Resource Management and Scheduling, January 2006.

[7] H. Casanova, A. Legrand, D. Zagorodnov, F. Berman. Heuristics for scheduling parameter sweep applications in Grid environments. *9th Heterogeneous Computing workshop*, 2000.

[8] M. Cannataro, D. Talia. The Knowledge Grid. *Communications of the ACM*, vol. 46, no. 1, pp. 89-93, 2003.

[9] H. Dail, H. Casanova, F. Berman. A modular scheduling approach for Grid application development environments. Submitted manuscript.

[10] X. Evers. A literature study on scheduling in distributed systems. Technical report, National Institute for Nuclear Physics and High-Energy Physics, Amsterdam, 92.

[11] M.R. Garey, D.S. Johnson. "Computers and intractability: a guide to the theory of NP-completeness". W.H. Freeman and Co.,1979.

[12] Global Grid Forum. *http://www.ggf.org/*

[13] GGF Scheduling and Resource Management Area. *http://www-unix.mcs.anl.gov/~schopf/ggf-sched/.*

[14] Global Grid Forum, "Defining the Grid: A Roadmap for OGSA(TM) Standards", H. Kishimoto, J. Treadwell (editors), Global Grid Forum Informational Document (GFD.53), 2005.

[15] Global Grid Forum, "Resource Management in OGSA", F. Maciel, J. Treadwell, L. Srinivasan, A. Westerinen, E. Stokes, H. Kreger, D. Snelling, Global Grid Forum Informational Document (GFD.45), 2004

[16] Global Grid Forum, "Web Services Agreement Specification", A. Andrieux, K. Czajkowski, A. Dan, K. Keahey, H. Ludwig, T. Nakata, J. Pruyne, J. Rofrano, S. Tuecke, M. Xu. , Global Grid Forum Informational Document (in public comment period), June 2006.

[17] Global Grid Forum, "Grid Scheduling Use Cases", P. Wieder, R. Yahyapour (editors), Global Grid Forum Informational Document (GFD.64), March 2006.

[18] The Globus Project. *http://www.globus.org/.*

[19] M. Iverson, F. Ozguner, and L. Potter. Statistical prediction of task execution times through analytic benchmarking for scheduling in a heterogeneous environment. In *Heterogeneous Computing Workshop*, 1999.

[20] A.A. Khan, C.L. McCreary, M.S. Jones. A comparison of multiprocessor scheduling heuristics. *International Conference on Parallel Processing*, 1994.

[21] K. Krauter, R. Buyya, M. Maheswaran. A taxonomy and survey of Grid resource management systems for distributed computing. *International Journal of Software: Practice and Experience*, Volume 32, Issue 2, 2002.

[22] Y. Kwok, I. Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys*, Vol. 31, n.4, 1999.

[23] C. Lee, D. Talia. Grid programming models: current tools, issues, and directions. In: F. Berman, G.C. Fox, A.J.G. Hey, "Grid computing – making the global infrastructure a reality", 2003.

[24] M. Mika, G. Waligora, J. Weglarz. A metaheuristic approach to scheduling workflow jobs on a Grid. In: J. Nabrzyski, J. Schopf, J. Weglarz (Eds.),*Grid Resource Management*, Kluwer, 2003.

[25] J. Nabrzyski, J. Schopf, J. Weglarz (Eds.). *Grid Resource Management*, Kluwer, 2003.

[26] S. Orlando, P. Palmerini, R. Perego, F. Silvestri. Scheduling high-performance data mining tasks on a data Grid environment. *Europar Conf.*, 2002.

[27] M. Pinedo. "Scheduling: Theory, Algorithms, and Systems". Prentice Hall, 2001.

[28] T.G. Robertazzi. Ten Reasons to Use Divisible Load Theory. *IEEE Computer*, Vol. 36, n. 5, 2003.

[29] N. Sample, P. Keyani, G. Wiederhold. Scheduling under uncertainty: planning for the ubiquitous Grid. *COORDINATION 2002*, LNCS 2315, 2002.

[30] H.J. Siegel, S. Ali. Techniques for mapping tasks to machines in heterogeneous computing systems. *Journal of Systems Architecture*, 46, 2000.

[31] H. Topcuoglu, S. Hariri, M. Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions on parallel and distributed systems*, Vol. 13, n.3, 2002.

[32] R. Wolski. Dynamically forecasting network performance using the Network Weather Service. *Cluster Computing*, Vol. 1, n. 1, 1998.

[33] R. Wolski, N. Spring, J. Hayes. Predicting the CPU availability of time-shared Unix systems. UCSD tech. rep., 1998.

[34] R. Wolski, N. Spring, J. Hayes. The network weather service: a distributed resource performance forecasting service for metacomputing. *Future Generation Computer Systems*, Vol. 15, 1999.

[35] A. Yarkhan, J.J. Dongarra. Experiments with scheduling using simulated annealing in a Grid Environment. *Grid Conf.*, 2002.