A PROPOSAL FOR A GENERIC GRID SCHEDULING ARCHITECTURE [*]

Nicola Tonellotto
Institute of Information Science and Technologies, 56100 Pisa, Italy
Information Engineering Department, University of Pisa, 56100 Pisa, Italy
nicola.tonellotto@isti.cnr.it

Ramin Yahyapour
Robotics Research Institute, University of Dortmund, 44221 Dortmund, Germany
ramin.yahyapour@udo.edu

Philipp Weider
Central Institute for Applied Mathematics, Research Centre Jülich, 52425 Jülich, Germany
ph.wieder@fz-juelich.de

Abstract In the past years, many Grids have been deployed and became commodity systems in production environments. While several Grid scheduling systems have already been implemented, they still provide only "ad hoc" and domain-specific solutions to the problem of scheduling resources in a Grid. However, no common and generic Grid scheduling system has emerged yet. In this work we identify generic features of three common Grid scheduling scenarios, and we introduce a single entity that we call scheduling instance that can be used as a building block for the scheduling solutions presented. We identify the behavior that a scheduling instance must exhibit in order to be composed with other instances to build Grid scheduling systems discussed, and their interactions with other Grid functionalities. This work can be used as a foundation for designing common Grid scheduling infrastructures.

Keywords: Grid computing, Resource management, Scheduling, Grid middleware.

[*]This paper includes work carried out jointly within the CoreGRID Network of Excellence funded by the European Commission's IST programme under grant #004265.

1. Introduction

The allocation and scheduling of applications on a set of heterogeneous, dynamically changing resources is a complex problem. There are still no common Grid scheduling strategies and systems available which serve all needs. The available implementations of scheduling systems depend on the specific architecture of the target computing platform and the application scenarios. The complexity of the applications and the user requirements on the one side and the system heterogeneity on the other don't permit to efficiently perform manually any scheduling procedure.

The task of scheduling applications does not only include the search for a suitable set of resources to run applications with regard to some user-dependent Quality of Service (QoS) requirements; moreover the scheduling system may be in charge of the coordination of time slots allocated on several different resources to run the application. In addition dynamic changes of the status of resources must be considered. It is the task of the scheduling system to take all those aspects into account to efficiently run an application. Moreover, the scheduling system must execute these activities while balancing several optimization functions: one provided by the user with her objectives (e.g. cost, response-time) as well as several other objectives represented by the resource providers (e.g. throughput, profit).

These problems increase the complexity of the allocation and scheduling problem. Note that Grid scheduling significantly differs from the conventional job scheduling on parallel computing system. Several Grid schedulers have been implemented in order to reduce the complexity of the problem for particular application scenarios. However, no common and generic Grid scheduler yet exists, and probably there will never be one as the particular scenarios will require dedicated scheduling strategies to run efficiently. Nevertheless several common aspects can be found in these existing Grid schedulers which lead to assumption that a generic architecture may be conceivable which not only simplifies the implementation of different scheduling but also provide an infrastructure for the interaction between these different systems. Ongoing work [7]in the Global Grid Forum is describing those common aspects, and starting from this analysis we propose a generic architecture describing how a generic grid scheduler should behave.

In Section 2 we analyze three common Grid scheduling scenarios, namely Enterprise Grids, High Performance Computing Grids and Global Grids. In Section 3 we identify the generic characteristics of the previous scenarios and their interactions with other Grid entities/services. In

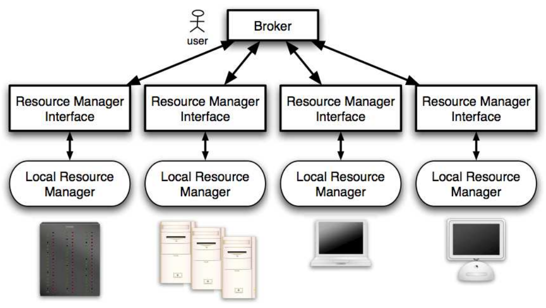
user
Broker
Resource Manager Interface
Resource Manager Interface
Resource Manager Interface
Resource Manager Interface
Local Resource Manager
Local Resource Manager
Local Resource Manager
Local Resource Manager

Figure 1. Example of a scheduling infrastructure for Enterprise Grids

Section 4 we introduce a single entity that we call scheduling instance that can be used as a building block for the scheduling architectures presented and we identify the behavior that this scheduling instance must exhibit in order to be composed with other instances to build the Grid scheduling systems discussed.

2. Grid Scheduling Scenarios

In this Section three common grid scheduling scenarios are briefly presented. This list is neither complete nor exhaustive. However, it represents common architectures that are currently implemented in application-specific Grid systems, either in research or commercial environments.

2.1 Scenario I: Enterprise Grids

Enterprise Grids represent a scenario of commercial interest in which the available IT resources within a company are better exploited and the administrative overhead is lowered by the employment of Grid technologies. The resources are typically not owned by different providers and are therefore not part of different administrative domains. In this scenario we typically have a centralized scheduling architecture; i.e. a central broker is the single access point to the whole infrastructure and manages directly the resource manager interfaces that interact directly with the local resource managers (see Figure 1). Every user must submit jobs to this centralized entity.

2.2 Scenario II: High Performance Computing Grids

High Performance Computing Grids represent a scenario in which different computing sites, e.g. scientific research labs, collaborate for

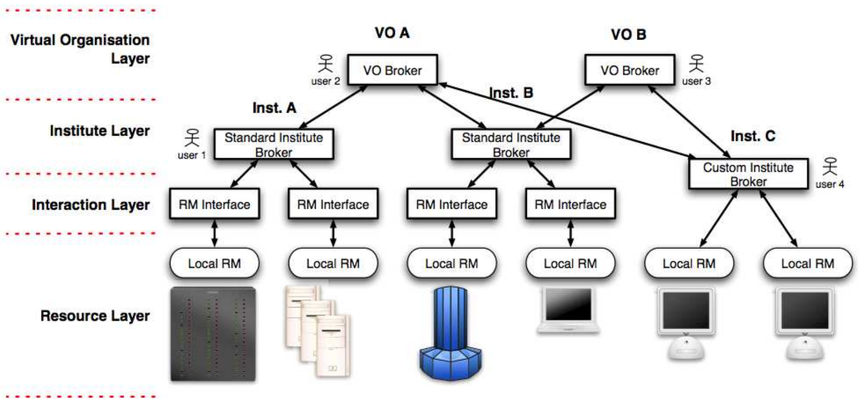

Figure 2. Example of a scheduling infrastructure for HPC Grids

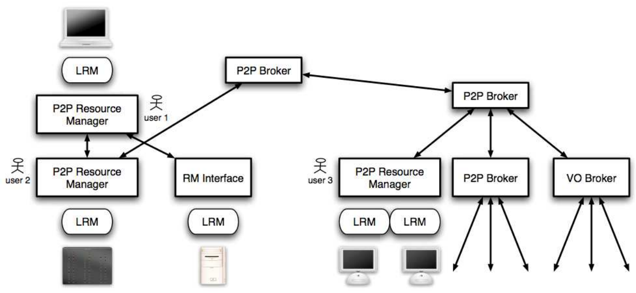

Figure 3. Example of a scheduling infrastructure for Global Grids

joint research. Here, compute- and/or data-intensive applications are executed on the participating HPC computing resources that are usually large parallel computers or cluster systems. In this case the resources are part of several administrative domains, with their own policies and rules.

A user can submit jobs to the broker at institute or VO level. The brokers can split a scheduling problem into several sub-problems, or forward the whole problem to different brokers in the same VO.

2.3 Scenario III: Global Grids

Global Grids might comprise all kinds of resources, from single desktop machines to large-scale HPC machines, which are connected through a global Grid network. This scenario is the most general one, covering both cases illustrated above and introducing a fully decentralised architecture. Every Peer-to-Peer broker can accept jobs to be scheduled, as Figure 3 depicts.

3. Common functions of Grid Scheduling

The three scenarios illustrated in the previous section show several entities interacting to perform scheduling. To solve scheduling problems, these entities can perform several tasks as described in [3–4]. To perform them, they can interact with other entities/services, both external ones and those part of the GSA implementation. Exploiting the information presented in [7, 5], it is possible to identify a detailed list of core independent functions that can be used to build specific Grid scheduling systems. In the following a list of atomic, self-contained functions is presented; these functions can be part of any complex mechanism or process implemented in a generic Grid Scheduling Architecture (GSA).

- **Naming**: Every entity in play must have a unique identifier for interaction and routing of messages. Some mechanism must be in charge of assigning and tracking unique identifiers to the involved entities.

- **Security**: Every interaction between different un-trusted entities may need several security mechanisms. A scheduling entity may need to certify its identity when contacting another scheduling instance, when it is trying to collect sensible information about other entities (e.g. planned schedules of other instances), or to discover what interactions it is authorized to initiate. Moreover, the information flow may need secure transport and data integrity guarantees, and a user may need to be authorized to submit a problem to a scheduling system. The security functions are orthogonal to other ones, in the sense that every service needs security-related mechanisms.

- **Problem Submission**: The entity implementing this function is responsible to receive a job to be scheduled from a user and submit it to a scheduling component. At this level, the definition of job is intentionally vague, because it depends on the particular job submitted (e.g. a bag of tasks, a single executable, a workflow, a DAG). The job to be scheduled is provided using a user-defined language, and must be translated into a common description that is shared by some scheduling components.This description will therefore be exploited in the whole scheduling process. It should be able to identify scheduling related terms and to build agreement templates used by the scheduling instances to schedule the job.

- **Schedule Report**: An entity implementing this function must receive the the answer of the scheduling instance to a previously submitted problem and translate it into a representation consumable by the user.
- **Information**: A scheduling instance must have coherent access to static and dynamic information about resources characteristics (computational, data, networks, etc.), resource usage records, job characteristics, and, in general, services involved in the scheduling process. Moreover, it must be able to publish and update its own static and dynamic attributes to make them available to other scheduling instances. These attributes include allocation properties, local scheduling strategies, negotiation mechanism, local agreement templates and resource information relevant to the scheduling process [4]. It can be in addition useful to provide the capability to cache historical information.
- **Search**: This function can be exploited to perform optimized information gathering on resources. For example, in large scale Grids it can be neither important nor efficient to collect information about every resource, but just a subset of "good" candidate resources. Several search strategies can be implemented (e.g. "best fit" searches, P2P searches with caching, iterative searches). Every search should include at least two parameters: the number of records requested in the reply and a time-out for the search procedure.
- **Monitoring:** A scheduling infrastructure can monitor different attributes to perform its functions: it can be useful to monitor e.g. the status of an agreement or an allocation to check if they are respected, the execution of a job to undertake next scheduling or corrective actions, or the status of a scheduling description through the whole system for user feedback.
- **Forecasting**: In order to calculate a schedule it can be useful to rely on forecasting services to predict the values of the quantities needed to apply a scheduling strategy. These forecasts can be based on historical records, actual and/or planned values.
- **Performance Evaluation**: The description of a job to be scheduled can miss some information needed by the system to apply a scheduling strategy. In this case it can be possible to exploit performance evaluation methodologies based on the available job description in order to predict the unknown information.

- **Reservation**: In order to schedule complex jobs as workflows and co-allocated tasks, as well as jobs with guarantees, it is in general necessary to reserve resources for particular time frames. The reservation of a resource can be obtained in several ways: automatically (because the local resource manager enforces it), on demand (only if explicitly requested from the user), etc. Moreover, the reservations can be restricted in time: for example only short-time reservations (i.e. with a finite time horizon) can be available. This function can require interaction with local resource managers and can be in charge of keeping information about allotted reservation and reserve new time frames on the resource(s).
- **Coallocation**: This function is in charge of the mechanisms needed to solve coallocation scheduling problems, in which strict constraints on the time frames of several reservations must be respected (e.g. the execution at the same time of two highly interacting tasks). It can rely on a low-level clock synchronization mechanism.
- **Planning**: When dealing with complex jobs (e.g. workflows) that need time-dependent access to and coordination of several objects like executables, data and network paths, a planning functionality, potentially built on top of a reservation service, is required.
- **Agreement**: In case quality of service guarantees concerning e.g. the allocation and execution time of a job must be considered, an agreement can be created and manipulated (e.g. accepted, rejected and modified) by the participating entities. A local resource manager can publish through its resource manager interface an agreement template regarding the jobs it can execute and a problem can include an agreement template regarding the guarantees that it is looking for.
- **Negotiation**: To reach an agreement the interacting partners may need to follow particular rules to exchange partial agreements to reach a final decision (e.g. who is in charge of providing the initial agreement template, who may modify what, etc.). This function should include a standard mechanism to implement several negotiation rules.
- **Execution**: This function is responsible to actually execute the scheduled jobs. It must interact with the local resource manager to perform the actions needed to run all the components of a job (e.g. staging, activation, execution, clean up). Usually it interacts with a monitoring system to control the status of the execution.

- **Banking**: The accounting/billing functionalities are performed by a banking system. It must provide interfaces to access accounting information, charging (in case of reservations or use of resources) and refunding (in case of agreement failures).

- **Translation**: The interaction with several services that can be implemented differently can force to translate information about the problem from the semantics of one system to the semantics of the other.

- **Data Management Access**: Data transfers can be included in the description of jobs. Although data management scheduling shows several similarities with job scheduling, it is considered a distinct, stand-alone functionality because the former shows significant differences compared to the latter (e.g. replica management and repository information) [2]. The implementation of a scheduling system can need access to data management facilities to program data transfers with respect to planned job allocations, data availability and eligible costs. This functionality can rely on previously mentioned ones, like information management, search, agreement and negotiation.

- **Network Management Access**: Data transfers as well as job interactions can need particular network resources to respect guarantees on their execution. As in the previous case, due to its nature and complexity, network management is considered a stand-alone functionality that should be exploited by scheduling systems if needed [1, 6]. This functionality can rely on previously mentioned ones, like information management, search, agreement and negotiation.

4. Scheduling Instance

It is possible to consider the different blocks in the previous examples as particular implementations of a more general software entity called scheduling instance. In this context, a scheduling instance is defined as a software entity that exhibits a standardized behavior with respect to the interactions with other software entities (which may be part of a GSA implementation or external services). Such scheduling entities cooperate to provide, if possible, a solution to scheduling problems submitted by users, e.g. the selection, planning and reservation of resource allocations for a job [4].

The scheduling instance is the basic building block of a scalable, modular architecture for scheduling tasks/jobs/workflows/applications in Grids. Its main function is to find a solution to a scheduling problem that it receives via a generic input interface. To do so, the scheduling instance needs to interact with local resource management systems that typically control the access to the resources. If a scheduling instance can find a solution for a submitted scheduling problem, the generated schedule is returned via a generic output interface.

From the examples depicted above it is possible to derive a high level model of operations for a generic set of cooperating scheduling instances. To provide a solution to a scheduling problem, a scheduling instance can exploit several options:

- It can try to solve the whole problem by itself interacting with local resource managers that it is able to interact with.
- If it can partition the problem in several sub-problems, it can try to:
  1. solve some of the sub-problems, if possible,
  2. negotiate to forward the unsolved sub-problems to independent scheduling instances,
  3. wait for potential solutions coming from other scheduling instances, or
  4. aggregate localized solutions to find a global solution for the original problem.
- If it cannot partition the problem or cannot find a solution by aggregating sub-problem solutions, it has two options:
  1. it can report back that it cannot find a solution or
  2. it can
     - negotiate to forward the whole problem to another, different scheduling instance or
     - wait for a solution to be delivered by the instance the problem has been forwarded to.

A generic Grid Scheduling Architecture will need to cover these behaviors, but actual implementations do not need to implement all of them. This model of operations is clearly modular, and permits to implement several scheduling infrastructures, like the ones depicted in the previous examples.

From them we can infer that a generic scheduling instance can exhibit the following abilities:

- interact with local resource managers;

- interact with external services that are not defined in the Grid Scheduling Architecture, like information, forecasting, submission, security or execution services;
- receive a scheduling problem (from other scheduling instances or external submission services), calculate a schedule, and return a scheduling decision (to the calling instance or an external service);
- split a problem in sub-problems, receive scheduling decisions and merge them into a new one;
- forward problems to other scheduling instances.

However, an instance might exhibit only a subset of such abilities. This depends on its interactions with other instances/services and its expected behavior (e.g. the ability to split and/or forward problems).

If a scheduling instance is able to cooperate with other instances, it must exhibit the ability to send problems or sub-problems, depending on the case, and receive scheduling results. Looking at such an instance, we call higher level instances the ones that are able to directly forward a problem to that instance, and lower level instances the ones that are able to directly accept a problem from that instance. A single instance must act as a decoupling entity between the actions performed at higher and lower levels: it is concerned neither with the previous instances through which the problem flows (i.e. it has been submitted by an external service or forwarded by other instances as a whole problem or as a sub-problem), nor with the actions that the following instances will undertake to solve the problem. Every instance will need to know just the problem it has to solve and the source of the original scheduling problem that helps to resolve, to avoid potential forwarding issues.

From a component point of view abilities as described above are expressed as interfaces. In general, the interfaces of a scheduling instance can be divided in two main categories: functional interfaces and non-functional interfaces. The former are necessary to enable the main behaviors of the scheduling instance, while the latter are concerned with the management of the instance itself (creation, destruction, status notification, etc.). We want to highlight that we considered only the functionalities that must be directly exploited to support a general scheduling architecture; for example, security services are from a functional point of view not strictly needed to schedule a job, so they are considered external services or non-functional interfaces. The functional interfaces that a scheduling instance can expose are depicted in Figure 4 and in detail described in the following:

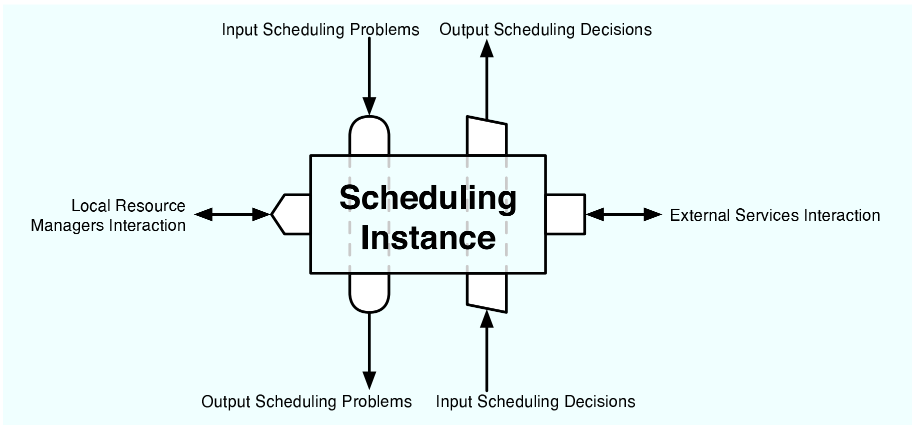
Input Scheduling Problems
Output Scheduling Decisions
Local Resource Managers Interaction
Scheduling Instance
External Services Interaction
Output Scheduling Problems
Input Scheduling Decisions

Figure 4. Functional interfaces of a scheduling instance

Input Scheduling Problems Interface The methods of this interface are responsible to receive a description of a scheduling problem that must be solved, and start the scheduling process. This interface is not intended to accept jobs directly from users; rather an external submission service (e.g. portal or command line interface) can collect the scheduling problems described with a user-defined formalism, validate them and produce a neutral representation accepted as input by this interface. In this way, this interface is fully decoupled from external interactions and can be exploited to compose several scheduling instance as in the examples illustrated above, where an instance can forward a problem or submit a sub-problem to other instances using this interface.

Every scheduling instance must implement this interface.

Output Scheduling Decisions Interface The methods of this interface are responsible to communicate the results of the scheduling process started earlier with a problem submission. Like the previous one, this interface is not intended to communicate the results directly to a user, rather to a visualization/reporting service. Again, we can exploit this decoupling in a modular way: if an instance received a submission from another one, it must use this interface to communicate the results to the submitting instance.

Every scheduling instance must implement this interface.

Output Scheduling Problems Interface If an instance is able to forward a whole problem or partial sub-problems to other scheduling instances, it needs the methods of this interface to submit the problem to lower level instances.

Input Scheduling Decisions Interface If an instance is able to submit problems to other instances, it must wait until a scheduling decision is produced from the one which the problem was submitted to. The methods of this interface are responsible for the communication of the scheduling results from lower level instances.

Local Resource Managers Interface The final goal of a scheduling process is to find an allocation of the jobs to the resources. It means that sooner or later the whole process has to interact with local resource managers to allocate the jobs to the resources. While some scheduling instances can be dedicated to the "routing" of the problems, others interact directly with local resource managers to find suitable schedules, and propagate the answers in a neutral representation back to the entity that submitted the scheduling problem. Different local resource managers can require different interaction interfaces.

External Services Interaction Interfaces If an instance must interact with an entity that is neither a local resource manager nor another scheduling instance, it needs an interface that permits to communicate with that external service that is exploited by the scheduling architecture. For example, some instances may need to gain access to information, billing, security and/or performance predictor services.

Different external services can require different interaction interfaces.

5. Conclusion

In this paper we discussed a general model for Grid scheduling. This model in based on a basic, modular component we called scheduling instance. Several scheduling instance implementations can be composed to build existing scheduling scenarios as well as new ones. The proposed model has no claim to be the most general one, but the authors consider this definition a good starting point to build a general Grid Scheduling Architecture that supports cooperation between different scheduling entities for arbitrary Grid resources. The future work will be directed towards further specifying the interaction of the Grid scheduling instance to other scheduling instances as well as to the other mentioned middleware services. The outcome of this work should yield a common Grid scheduling architecture that allows the integration of several different scheduling instances that can interact with eachother as well as be exchanged for domain-specific implementations.

References

[1] V. Sander (Ed.). Networking Issues for Grid Infrastructure. GGF Document Series (GFD.37), 2004.

[2] R. W. Moore. Operations for Access, Management, and Transport at Remote Sites. GGF Document Series (GFD.46), 2005.

[3] J. M. Schopf. Ten Actions When Superscheduling. GGF Document Series (GFD.4), 2001.

[4] U. Schwiegelshohn and R. Yahyapour. Attributes for Communication between Scheduling Instances. GGF Document Series (GFD.6), 2001.

[5] U. Schwiegelshohn, R. Yahyapour, and Ph. Wieder. Resource management for future generation grids. Technical Report TR-0005, Institute on Scheduling and Resource Management, CoreGRID - Network of Excellence, May 2005.

[6] D. Simeonidou and R. Nejabati (Eds.). Optical Network Infrastructure for Grid. GGF Document Series (GFD.36), 2004.

[7] R. Yahyapour and Ph. Wieder (Eds.). Grid Scheduling Use Cases v1.2. GGF-GSA Working Draft, 2005.