GFD-I.064
Category: Informational
Grid Scheduling Architecture Research Group (GSA-RG)

Editors:
Ramin Yahyapour,
Philipp Wieder
March 26, 2006

Grid Scheduling Use Cases

Status of this Memo

This document provides information to the community regarding different Grid scheduling use case scenarios used in the definition of a Grid Scheduling Architecture. Distribution of this document is unlimited.

Copyright Notice

Abstract

Grids will provide a large variety of complex services. The interaction between those services requires resource management and scheduling solutions which allow the coordinated usage of the services, something which is currently not readily available.

Access to resources is typically subject to individual access, accounting, priority, and security policies imposed by the resource owners. In addition the consideration of different policies is also important for the implementation of various services like for example accounting or billing services. Generally those policies are enforced by local management systems. Therefore an architecture that supports the interaction of independent local management systems with higher-level scheduling services is an important component for Grids. Further, user of a Grid may also establish individual scheduling objectives. Future Grid scheduling and resource management systems must consider those constraints in the scheduling process.

The goal of the Grid Scheduling Architecture Research Group (GSA-RG [1]) is to define a scheduling architecture that supports the cooperation between different scheduling instances for arbitrary Grid resources, including network, software, data, storage, and processing units. The research group will particularly address the interaction between resource management and data management. Co-allocation and the reservation of resources are key aspects of the new scheduling architecture, which will also integrate user- or provider-defined scheduling policies.

The group started with identifying a set of relevant use cases based on experiences obtained by existing Grid projects. These use cases are reported in this document. In a future document the GSA-RG will determine the required components of a generic and modular scheduling architecture and their interactions.

Contents

1 Introduction 5
2 Use cases 6
2.1 Scheduling complex workflows 6
2.1.1 Summary 6
2.1.2 Customers 6
2.1.3 Scenarios 6
2.1.4 Involved resources 7
2.1.5 Functional requirements 7
2.1.6 Workflow of scheduling process 8
2.1.7 Involved scheduling components/services 9
2.1.8 Failure considerations 10
2.1.9 Security considerations 10
2.1.10 Accounting considerations 10
2.1.11 Performance considerations 11
2.1.12 Use case situation analysis 11
2.2 Scheduling component-based applications for high performance heterogeneous computing 12
2.2.1 Summary 12
2.2.2 Customers 12
2.2.3 Scenarios 12
2.2.4 Involved resources 12
2.2.5 Functional requirements 13
2.2.6 Workflow of scheduling process 13
2.2.7 Involved scheduling components/services 14
2.2.8 Failure considerations 14
2.2.9 Security considerations 15
2.2.10 Accounting considerations 15
2.2.11 Performance considerations 15
2.3 Application-oriented scheduling in the KNOWLEDGE GRID 16
2.3.1 Summary 16
2.3.2 Customers 17
2.3.3 Scenarios 17
2.3.4 Involved resources 18
2.3.5 Functional requirements 18
2.3.6 Workflow of scheduling process 18
2.3.7 Involved scheduling components/services 18
2.3.8 Failure considerations 19
2.3.9 Security considerations 19
2.3.10 Accounting considerations 19
2.3.11 Performance considerations 19
2.3.12 Use case situation analysis 19
2.4 GRASP (Grid Resource Allocation Services Package) 20
2.4.1 Summary 20
2.4.2 Customers 21
2.4.3 Scenarios 21
2.4.4 Involved resources 22
2.4.5 Functional requirements 22
2.4.6 Workflow of scheduling process 23

2.4.7 Involved scheduling components/services 23
2.4.8 Failure considerations 24
2.4.9 Security considerations..24
2.4.10 Accounting considerations 24
2.4.11 Performance considerations...24
2.4.12 Use case situation analysis 24
2.5 Scheduling in loosely-coupled Grids with GridWay.................................25
2.5.1 Summary..25
2.5.2 Customers 25
2.5.3 Scenarios..25
2.5.4 Involved resources 26
2.5.5 Functional requirements..26
2.5.6 Workflow of scheduling process 27
2.5.7 Involved scheduling components/services 28
2.5.8 Failure considerations 32
2.5.9 Security considerations..32
2.5.10 Accounting considerations 32
2.5.11 Performance considerations...32
2.5.12 Use case situation analysis 33
2.6 Agent-based scheduling with Calana..34
2.6.1 Summary..34
2.6.2 Customers 35
2.6.3 Scenarios..35
2.6.4 Involved resources 35
2.6.5 Functional requirements..35
2.6.6 Workflow of scheduling process 36
2.6.7 Involved scheduling components/services 36
2.6.8 Failure considerations 36
2.6.9 Security considerations..36
2.6.10 Accounting considerations 37
2.6.11 Performance considerations...37
2.6.12 Use case situation analysis 37
2.7 Meta-scheduling/co-allocation in the VIOLA testbed...............................38
2.7.1 Summary..38
2.7.2 Customers 40
2.7.3 Scenarios..40
2.7.4 Involved resources 40
2.7.5 Functional requirements..40
2.7.6 Workflow of scheduling process 40
2.7.7 Involved scheduling components/services 41
2.7.8 Failure considerations 41
2.7.9 Security considerations..41
2.7.10 Accounting considerations 41
2.7.11 Performance considerations...41
2.7.12 Use case situation analysis 42
3 Usage patterns 43
3.1 Simple job submission 43
3.2 Single-site execution with service guarantees..43
3.3 Job execution with advance reservation...44
3.4 Co-allocation 44

3.5 Complex workflow 44
3.6 Data Staging 45
4 References 46
5 Editor Information 48
6 Full Copyright Notice 50
7 Intellectual Property Statement 51

1 Introduction

One of the first milestones of the GSA-RG's charter is the identification of relevant use cases for Grid scheduling. This document is a collection of use case scenarios and profiles contributed by GSA-RG participants or solicited from others.

Based on this document the GSA-RG will identify and specify common requirements to support the creation of Grid schedulers which fulfil the requirements of these use cases. This information will be used to identify components, services and protocols for a Grid scheduling architecture and outline their interaction. Services and protocols from other GGF groups are considered as potential basic building blocks of such an architecture and will be used wherever possible.

Please note that it is not the task of this research group to define protocols or algorithms. Instead, the group identifies the requirements for Grid scheduling and designs a suitable Grid scheduling architecture by using existing services, protocols, etc. and by identifying currently missing components.

Section 2 comprises seven use cases. The first two of these cover abstract scenarios and the resulting requirements while the remaining five rely on scheduling and resource management solutions which have been developed in different projects. The structure of each sub-section describing one use case is identical to support the comparison and categorisation of the different use cases. Section 3 then lists a (not exhaustive) number of patterns one can determine when dealing with Grid scheduling and resource management. In Sections 4 to 7 references, editor information, the full copyright notice, and the intellectual property statement are given.

2 Use cases

2.1 Scheduling complex workflows

2.1.1 Summary

Many Grid applications require the coordinated processing of complex workflows which includes scheduling of heterogeneous resources within different administrative domains. A typical scenario is the coordinated scheduling of computational resources in conjunction with data, storage, network and other available Grid resources, like software licenses, experimental devices, etc. The Grid scheduler should be able to coordinate and plan the workflow execution. That is, it should reserve the required resources and create a complete schedule for the whole workflow in advance.

In addition, cost management and accounting have to be considered in the scheduling process.

2.1.2 Customers

This use case is of interest for a wide variety of costumers namely every Grid user who wants to process complex workflows. For instance, the presented use case is common in climate-research and high-energy physics.

2.1.3 Scenarios

Since this use case defines the general requirements for scheduling complex workflows a broad variety of scenarios is possible. This includes the "classical" example of scheduling a computational job including network, data, software and storage, but also covers examples like Grid based steering of simulations or experiments.

A typical example would be the following user request, where

- 48 processing nodes of a specific type,
- 1 GB of available memory, and
- a specific licensed software package

are requested for 1 hour between 8am and 6pm of the following day. In addition, a specific visualization device should be available during program execution, which requires a minimum bandwidth between the visualization device and the main computer during program execution. Furthermore

- the program relies on a specific input data-set from a data repository,
- the user wants to spend at most 5 Euro, and
- he prefers a cheaper workflow execution to an earlier execution.

A Grid scheduler should be able to generate a complete schedule for the execution of this workflow including all resources required for implicit actions before and after the actual workflow start for data management. However, this example should be considered as a quite simple scenario. In a real application it could easily be extended to contain additional workflow steps. The Grid scheduler should take the allocation of all requested resource types into account and, if necessary, should create advance reservations. Figure 2.1-1 shows an example of a possible scheduling output of a Grid scheduler.

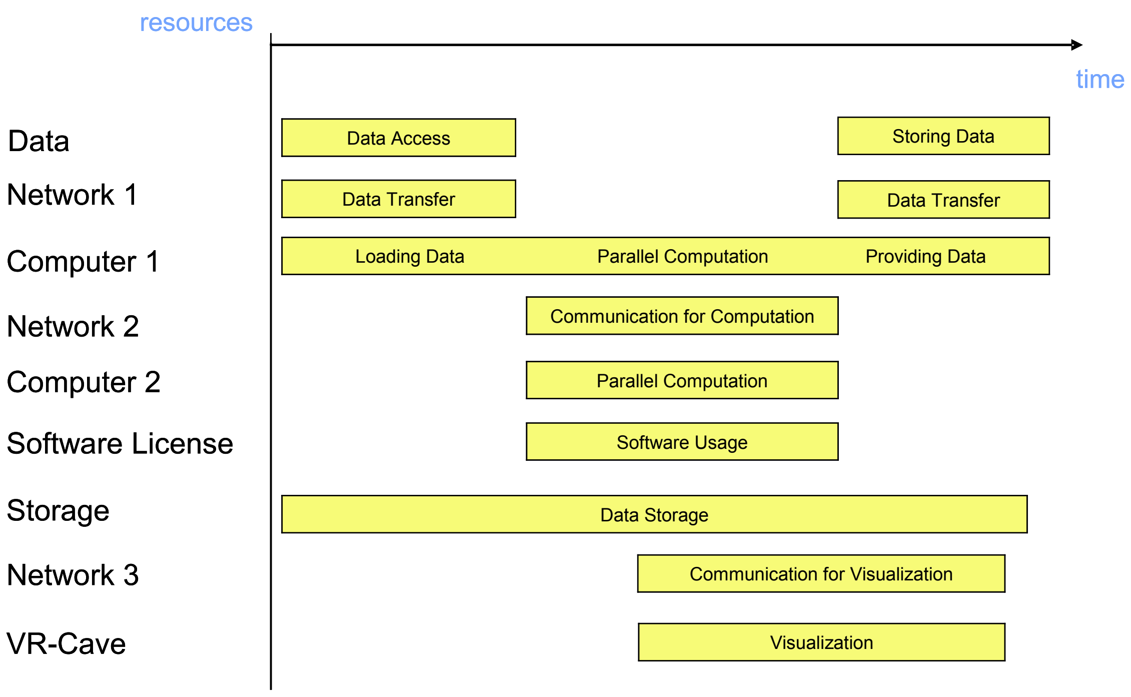

Figure 2.1-1 Example schedule

2.1.4 Involved resources

All kinds of available resources may be requested by the user, as long as the necessary means are in place to integrate them into the scheduling process. Figure 2.1-1 shows the usage of resources such as computing, data, storage, network and software resources, as well as special devices. But it can also be anticipated that services, sensors or even humans may be treated as resources in a Grid scheduling context.

2.1.5 Functional requirements

1. **Authentication, authorization, user right delegation & workflow integrity verification**. Authentication and authorization are essential for every Grid based workflow submission scenario. To enable the scheduler to act on behalf of the user the respective rights have to be delegated from the user to the scheduler. This use case also requires that the integrity of a workflow (or parts of the workflow) can be verified anytime during the scheduling process.
2. **Workflow parsing & validation**. The workflow description has to be parsed and formally validated (workflow pre-processing).
3. **Information retrieval (static & dynamic)**. To map the resource requests contained in the workflow description onto available resources, information about the resources and their status has to be retrieved from appropriate entities (and offered by these entities). It should be possible to gather static[1] and dynamic resource information separately to restrict the time-consuming dynamic information retrieval.

[1] Information is called static if it is known to be valid after the job has terminated. This may be the case e.g. concerning certain software available on a system, the maximum number of CPUs of a compute cluster, etc.

4. **Resource pre-selection**. To avoid information queries on resources which do not fulfil policy constraints defined by the user or which are definitely not capable of fulfilling a resource request (why should one ask for information about the current system state if the system has less processors than required by the user), a set of resources should be selected based on those so-called "static" resource information.
5. **Service choreography, management**. It might be useful to have mechanisms which allow choreography/management of services which represent the pre-selected resources on different levels to obtain the desired dynamic information faster and more reliable (see 2.1.6 for the chronology of the scheduling process).
6. **Scheduling**. A schedule has to be generated based on the information about the workflow and the resources, accounts, etc.
7. **Advance reservation/agreement negotiation**. It is essential to meet time or precedence requirements defined by the workflow. Therefore one has to reserve in advance the resources selected by the schedule to guarantee the proper execution of the workflow. One approach to achieve this is specified by the GRAAP-WG [2], called Web Services Agreement. This specification defines a language/protocol to negotiate agreements between service provider and consumer.
8. **Workflow execution/processing**. The workflow has to be processed. It is assumed that the local resource managers execute the atomic entities a workflow consists of. To process the workflow or parts of it, a workflow engine or processor is needed.
9. **Accounting**. Information about the status of a workflow during its lifetime and the resources needed to execute the workflow is collected. In this case such information is especially needed to correctly bill for the enlisted services.
10. **Billing**. According to the service-level negotiated and the one retrieved users may be charged for services they have obtained. Price, penalties etc. are subject to negotiation between resource provider and resource consumer.
11. **Failure management**. This is essential not only to have an instrument to monitor and possibly re-schedule workflows in case of failure within the system, but also to provide users with information and tools to manage such failure situations.

2.1.6 Workflow of scheduling process

The different steps of the scheduling process are described in this section referring to the example introduced in Section 2.1.3 (For each step the services needed are listed in brackets, see Section 2.1.7).

1. **Composition and submission of the workflow request**. The workflow description is generated and transferred to an entity capable of processing its contents. In case of the example a workflow will be generated that contains the resource requests and constraints listed in Section 2.1.3. With respect to this use case no specific language to describe the workflow request is demanded. *(Services 1 and 2)*
2. **Pre-processing of the workflow request**. The workflow request has to be parsed and validated if possible. If the entity pre-processing the workflow is unable to do so it may try to translate the workflow to a suitable description. *(Services 2 and 5)*

3. **Gathering of static resource information**. Some service is needed which gathers static information about existing resources. This service may be an information service or a database. It is also possible that some Web Service Resource Property [3] is queried to gather static information about the service. Concerning the example it is assumed that this processing step identifies a pool of 800 resources of all requested kinds. *(Services 3, 4, 8 and 9)*
4. **Pre-selection of resources**. Based on the information collected in Step 3. algorithms are used to limit the number of resources which are potentially capable of participating in the workflow's processing. With regard to the example this may cut down resource candidates to 30, since e.g. some systems may not have 48 processors, may not offer the software requested or the respective system is maintained the next day. *(Service 3)*
5. **Query of dynamic resource information**. The dynamic query delivers information like whether the current load of the machine allows allocating 48 processors (this is different from Step 4., where resources are sorted out because they offer less than 48 processors). This again limits the number of potential resources which are actually used in the next step to process the schedule. *(Services 3, 4, 8 and 9)*
6. **Generation of schedule and initialization of required reservations**. Based on the resource information gathered in the previous steps a schedule is generated (e.g. as shown in Figure 2.1-1). It is then attempted to reserve the necessary resources in advance, a process which may fail several time due to the complexity of the workflow and the number of dependencies between the reservations needed. A failed negotiation may lead to re-scheduling possibly with a preceding step 5. *(Services 2 and 6)*
7. **Execution of workflow**. Once the schedule as shown in Figure 2.1-1 is confirmed it is processed and executed. In case of the example at first data is taken from some storage system and transferred via network 1 to computer 1. If no error occurs the workflow is executed until the last chunk of resulting data is written via network 1 to storage. *(Services 2 and 7)*
8. **Completion of workflow**. This includes the finalization of accounting and billing as well as the delivery of the data the workflow produced. *(Services 1, 2, 8 and 9)*

2.1.7 Involved scheduling components/services

The following services are required (Please note that this does not imply a separate service implementation for every entity listed here. The term service is used in the sense of some functionality provided by a certain software component, which may integrate several services. For each service the scheduling process steps it is involved in are listed in brackets, see Section 2.1.6):

1. User or an agent acting on-behalf of a user. *(Scheduling process steps 1 and 8)* The user/agent may also be involved in adjustments of the workflow if the systems permit that. This may happen at different steps, e.g. due to some failure condition.
2. Scheduling and resource management service. *(Scheduling process steps 1, 2, 3, 6, 7 and 8)*
3. Brokering service. *(Scheduling process steps 3, 4 and 5)*
4. Information service. *(Scheduling process steps 3 and 5)*
5. Translation service. *(Scheduling process step 2)*
6. Negotiation service. *(Scheduling process step 6)*

7. Execution service. *(Scheduling process step 7)*
8. Accounting service. *(Scheduling process steps 3, 5 and 8)*
9. Billing service. *(Scheduling process steps 3, 5 and 8)*

2.1.8 Failure considerations

Based on 2.1.6 the following failures have to be taken into consideration:

- *(Processing of the workflow request)*
  1. The parser does not support the format of the workflow.
  2. The workflow request is not valid.
- *(Gathering of static resource information)*
  3. The information source(s) needed to gather static information are not available.
- *(Pre-selection of resources)*
  4. The pre-selection of resources prevents workflow from being executed since some resource request(s) already cannot be fulfilled.
- *(Query dynamic resource information)*
  5. The information source(s) needed to query dynamic information are not available.
- *(Generation of schedule & initialization of required reservations)*
  6. Requested resources are not available. The result of the dynamic resource query indicates that one or many of the resources requested are not available (maybe due to local resource manager failures, etc.).
  7. Precedence relations/time constraints cannot be met. The initialization of reservations required by the schedule fails for one/many resources.
  8. Time out. No schedule could be generated within a pre-defined timeframe.
- *(Execution of workflow)*
  9. The execution of the workflow may fail for different reasons like e.g. temporary system unavailability, unrecoverable errors in the user code, etc.

Failures like unavailability of services, network, etc. are not considered here since those are use case independent failures.

2.1.9 Security considerations

The functional requirements section lists the four most prominent security features demanded by this use case (see Section 2.1.5, bullet 1.). In general it has to be noted that protection of the user's identity, the workflow's integrity and the confidentiality of information has to be guaranteed throughout the whole process described here.

2.1.10 Accounting considerations

- **Local domain accounting**. The use case described here does not define any demands concerning additional accounting mechanisms in addition to what is already implemented locally. But accounting information provided by the local resource administrators may have implications on the scheduling decisions so that e.g. specific resources are not available due to temporary local restrictions. To consider this information in the scheduling process it has to be available through the extended information service/broker (which implies an appropriate interface).
- **Inter-domain accounting**. The accounting/billing service is in the light of this use case a black box providing interfaces to send/receive accounting/billing information. Of greater interest are the information itself and the resulting brokering/scheduling decisions as well as the integration of an

accounting/billing system into the system derived from this use case. It is suggested to refer to other activities at GGF (like RUS [4] and work carried out in projects.

2.1.11 Performance considerations

The main impact on the performance of the whole process as described in Section 2.1.6 has the communication between the involved components/services. This includes the following items:

- **Scalability**. If the amount of resources which are part of a Grid increases, the communication between local resource managers and the scheduling service or the extended information service may have a negative impact on the overall system performance. Solutions like information caching (e.g. based on WS-Notification [5]) may be applied.
- **Choice of the service programming model**. Assuming that instances of that use case are performed in a Web Service based environment using SOAP [6] to exchange messages, one has to be aware that the performance is in general seen to be worse than that of other solutions like e.g. CORBA [7].
- **Communication failure**. In a service-oriented architecture as described above the failure of communication between services is not unusual. To realize a reliable system and enforce a certain level of service quality (and therefore increase performance), mechanisms are needed to manage services. One activity which is to be monitored here is the Web Services Distributed Management TC [8].

The performance impact of the resource request to resource offer mapping and the schedule generation is highly influenced by the performance of the implemented algorithms, but also by the estimated number of involved resources.

2.1.12 Use case situation analysis

Many research and development activities are underway to find solutions for scheduling complex workflows as described in this use case, but no consistent and broadly applicable solution is available yet. It is envisaged that the Grid Scheduling Architecture Research Group will define an architecture which, once implemented, will provide the functions required by this use case.
It is of particular interest that the scheduling architecture derived from this (and other) uses case(s) is as much independent from the resource types involved as possible.

2.2 Scheduling component-based applications for high performance heterogeneous computing

2.2.1 Summary

Component-based programming is currently a promising paradigm for programming complex systems. By breaking complex applications into smaller and simpler pieces; component-based programming is well-suited to efficiently face new challenges in terms of programmability, interoperability, code reuse and efficiency that mainly derive from the features that are typical for Grids [9].
Such applications may require access to several kinds of resources for their execution, mainly computing and network ones, but the availability of storage resources can also be required.
One of the main requirements to be taken into account when scheduling these applications is co-scheduling which is imposed by the strong interactions between the different application's components.

2.2.2 Customers

This use case is of interest for a wide variety of costumers that use specific programming environments (e.g. ASSIST [10][11], GridCCM [12], or ProActive [13]) to write, compile, deploy, and execute component-based applications. For example, ASSIST is used to build and execute several HPC applications in the areas of Computer Graphics, Data Mining, Earth Observations, etc.

2.2.3 Scenarios

A component-based application can be very generic; in most cases a stream-like online communication pattern is adopted as the interaction mechanism between high performance components. The basic requirements for this kind of applications are high computational power and high communication bandwidth. This means that every single resource that can influence these two quantities has to be involved in the scheduling process.
To exploit stream-based communications, it is important to guarantee the co-allocation of the resources involved, which implies some sort of reservation mechanism to manage immediate and future co-allocations.
Moreover, the dynamic behaviour of the Grid imposes the need of adaptation mechanisms for this kind of applications, i.e. the application has to be monitored at runtime and possible re-scheduling actions must be undertaken to guarantee a certain level of service. This implies that it is necessary to adapt dynamically the current resource allocations for components which do not perform on a given set of resources according to their guaranteed service-level.

2.2.4 Involved resources

The experience with the ASSIST programming environment has shown that most resources involved in the execution of HPC applications are computing resources, network paths (guaranteed sockets and bandwidths) and storage access rights. With the introduction of hosting environments for the components, access to the information about available components, deployment mechanisms for new components and configuration procedures for existing ones can be considered to be resources that we also may have to deal with.

2.2.5 Functional requirements

1. **Authentication, authorization, user right delegation & job integrity verification**. Authentication and authorization are essential for every Grid based job submission scenario. To enable the scheduler to act on behalf of the user the respective rights have to be delegated from the user to the scheduler. This use case also requires that the integrity of a job (or parts of the job) can be verified anytime during the scheduling process, e.g. in the case of rescheduling.
2. **Job parsing & validation.** The job description has to be parsed and formally validated (job pre-processing). Moreover a performance model, coupled with user-level QoS requirements, needs to be built and evaluated.
3. **Information retrieval (static & dynamic)**. To map the resource requests contained in the job description onto available resources, information about the resources and their status has to be retrieved from appropriate entities (and offered by these entities). It should be possible to gather static ("static" with respect to the runtime of the job) and dynamic resource information separately in order to guarantee updated status information for the monitoring and rescheduling.
4. **Resource pre-selection.** To avoid information queries on resources which do not fulfil performance constraints defined by the user a set of resources should be selected based on the static resource information and the performance values.
5. **Co-allocation**. To find a suitable mapping of an application to resources the co-allocation of several heterogeneous resources has to be managed. For example, computational resources may be searched to allocate new components, reserve existing components for the execution and reserve high performance data pathways between several components in order to fulfil the QoS requirements.
6. **Scheduling**. A schedule has to be generated based on the information about the job and the resources, accounts, etc.
7. **Application execution**. The application has to be processed. It is assumed that the local resource managers/hosting environments execute the single components.
8. **Failure management**. During the execution the application has to be monitored to detect QoS violations, and, if possible, modify the current allocation/schedule to find a new solution that, with minimal modifications of the original schedule, can fulfil the QoS requirements.

2.2.6 Workflow of scheduling process

1. **Search for existing components**. To build an application, existing components can be re-used. The user can search for existing components with well-defined interfaces to build the application he wants to execute. The interfaces must be compatible in order to be able to compose new components.
2. **Composition of the application description and performance evaluation**. When all components needed by an application are known (by direct compilation or successful search) the application has to be described using a suitable language and its performance model has to be compiled and evaluated in order to drive the resource selection, orchestration and scheduling phases.

3. **Gathering of resource information**. Some service is needed which gathers static information about existing resources. This service may be an information service or a database.
4. **Selection of single resources**. Based on the information collected in the previous step, algorithms are used to select resources which are potentially capable of participating in the application execution. These resources can be computational ones, as well as network links and hosting environments.
5. **Co-allocation of selected resources**. The single resources found in the previous phase have to cooperate at the same time to execute the application. If such cooperation (i.e. co-allocation with performance guarantees) can not be found, a new gathering and selection process must be undertaken.
6. **Generation of schedule and initialization of required reservations**. Based on the resource information gathered in the previous steps a schedule is generated. It may be necessary to reserve the necessary resource if the execution is not immediately possible. The reservation process may fail and that may lead to restart the selection scheduling phases at step 3.
7. **Launch of application**. At this point the existing components must be instantiated or activated and new ones must be transferred and activated on the selected resources.
8. **Monitoring**. An application monitoring facility may need to face situations in which the resource power required by the components/application changes significantly. It can use the components adaptation mechanisms, which may result in the activation of new components or the migration of existing ones.
9. **Re-scheduling**. If a decrease of the performance of an application forces an adaptation to a new execution state, it is necessary to find new resources that can cooperate with the existing ones to fulfil the user-level performance requirements and to schedule them for immediate execution.
10. **Completion of execution**. At the end of an application's execution, the components may need to be deactivated or removed from the hosting environments and the output data has to be sent back to the user.

2.2.7 Involved scheduling components/services

For each service the scheduling process steps it is involved in are listed in brackets, see Section 2.1.6):

1. Components search/composer service. *(Scheduling process steps 1 and 2)*
2. Performance evaluation service. *(Scheduling process steps 2 and 3)*
3. Static information service. *(Scheduling process step 3)*
4. Brokering service. *(Scheduling process steps 4 and 5)*
5. Translator service. *(Scheduling process steps 2 and 7)*
6. Co-allocation/Reservation service. *(Scheduling process step 6)*
7. Execution service. *(Scheduling process steps 7 and 10)*
8. Monitoring service. *(Scheduling process steps 8 and 9)*

2.2.8 Failure considerations

The following failures have to be taken into consideration:

- *(Search for existing components)*
  1. Failure to find requested components.
- *(Composition of the application description and performance evaluation)*
  2. Failure to build new components from the existing ones.
  3. Failure to build the performance model.

4. Failure to evaluate the performance model.

- *(Gathering of resource information)*
  5. The information source(s) needed to gather static information are not available.
- *(Selection of the single resources)*
  6. The selection algorithm fails to find the resources needed to fulfil the performance constraints.
- *(Co-allocation of the selected resources)*
  7. The co-allocation algorithm fails to find a mapping solution with the resources selected in the previous phase.
  8. Timeout. No co-allocation can be realised within a pre-defined timeframe.
- *(Generation of the schedule and initialization of required reservations)*
  9. Requested resources are not available for scheduling.
  10. The co-allocation algorithm fails to find an immediate or postponed co-allocation of the selected resources.
  11. Time constraints due to expected performance can not be met.
  12. Timeout. No schedule can be generated within a pre-defined timeframe.
- *(Launch of the application)*
  13. Errors in user-provided code.
  14. Errors in third-party components (code or faulty interface description).
- *(Completion of execution)*
  15. It is impossible to send back the results to the user.

2.2.9 Security considerations

The common security features required by most Grid scenarios (authorization, authentication, information integrity and confidentiality) are demanded by this use case as well. These features must be present in every service involved in the scheduling process, as well as in the external services. The security features demanded by the components used to build an application must be taken into account, but they are not part of the scheduling process.

2.2.10 Accounting considerations

Obviously, an accounting mechanism may be easily integrated into the scheduling scenario discussed, but accounting components are not of highest priority.

2.2.11 Performance considerations

Basically, the performance constraints of this use case are the same as those of the "Scheduling complex workflows" use case (see Section 2.1.11).

2.3 Application-oriented scheduling in the KNOWLEDGE GRID

2.3.1 Summary

The KNOWLEDGE GRID (K-GRID) is an architecture built atop basic Grid middleware services that defines more specific services for the definition, composition, validation and execution of knowledge discovery applications over Grids, and for storing and managing discovered knowledge [14][15]. The K-GRID *Resource Allocation and Execution Management Service* (*RAEMS*) is a service used by the KNOWLEDGE GRID to map applications onto available resources and to coordinate their execution. The K-GRID scheduler is part of the RAEMS; it can be seen as an "application agent" associated with each application to be executed. Indeed, the scheduler produces job assignments (along with timing constraints) for each application, with the goal of improving its performances, on the basis of knowledge or prediction about computational and I/O costs. Afterwards, it follows each application execution to adapt generated schedules to new information about job status and available resources. Moreover, since in realistic Grid applications it is generally infeasible to specify all the details of applications at composition time, the KNOWLEDGE GRID scheduler allows the definition and use of *abstract hosts*, i.e. hosts whose characteristics are only partially known, and that can be matched to different concrete ones [16].

Therefore, the main objectives of the scheduler are:

- *Abstraction from computational and network resources in application composition*. With the use of abstract hosts, users are allowed to disregard low-level execution-related aspects, and to concentrate more on the structure of their applications.
- *Application performance improvement*. Given a set of available hosts, schedules are generated trying to minimize applications' completion times.

Besides the scheduler, the K-Grid's RAEMS includes an Execution Manager, used to translate the output of the scheduling process into submissions to basic Grid services, and a Job Monitor that follows the execution of submitted jobs and notifies the scheduler about significant events occurred. Each K-Grid node has its own scheduler which is responsible for instantiating a new application agent for each scheduling request coming from the same or different nodes.

The architecture of the scheduler [17] comprises three main components (see Figure 2.3-1):

- *Mapper*. It computes schedules employing a scheduling algorithm and making use of resource descriptions and computational and I/O cost evaluations.
- *Cost/Size Estimator*. It builds the I/O and computational cost estimation functions. The CSE comprises *data gathering* modules collecting dynamic information about current and future availability and performance of resources, and *estimation modules* dealing with the actual construction of estimation functions, on the basis of the perceived status of resources with respect to time.
- *Controller*. It guides the scheduling activity by receiving abstract applications, requesting the corresponding schedules to the Mapper, and ordering the execution of scheduled jobs to the Execution Manager. The Controller also

receives notifications about significant events occurred and re-schedules unexecuted parts of the application.

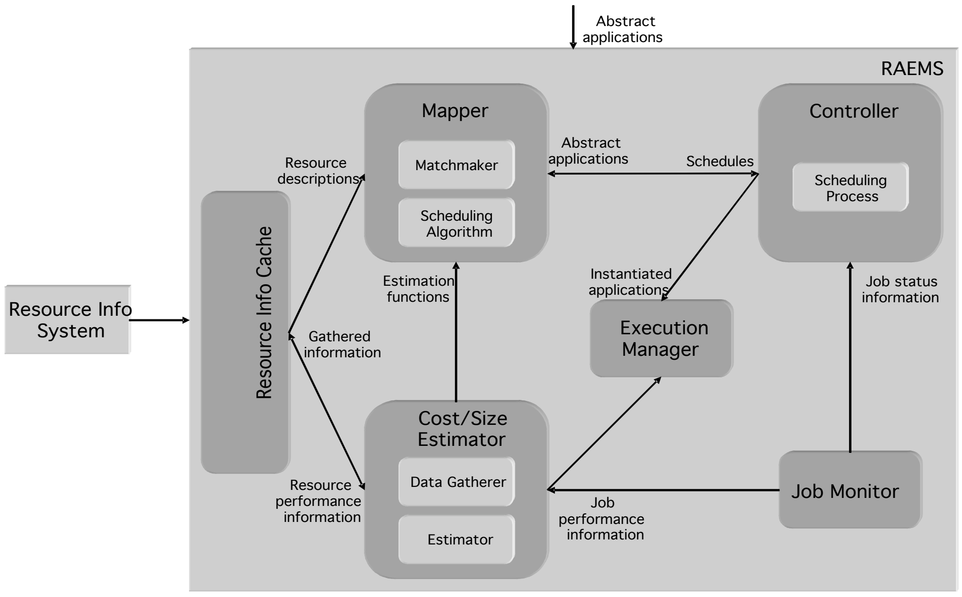

Figure 2.3-1 K-GRID scheduler architecture

The scheduler modules are extensible as they provide an open interface allowing to plug in user-defined functionalities and behaviours. The scheduler can load modules implementing scheduling algorithms and matchmaking functionalities (in the Mapper), scheduling processes (in the Controller), and data gathering and cost estimation activities (in the Cost/Size Estimator). Each module can refer to its own description of resources. This makes the scheduler potentially useful in Grid frameworks different from the KNOWLEDGE GRID. For instance, cooperation among different schedulers could be implemented in the scheduling process, and resource and applications' descriptions could be properly designed to include the needed information.

2.3.2 Customers

The target customers of the KNOWLEDGE GRID scheduler are mostly Grid users who want to perform knowledge discovery processes on Grids. However, since the scheduler is not tightly coupled with the KNOWLEDGE GRID architecture, its use can be seamlessly extended to other application domains.

2.3.3 Scenarios

- **Application submission**: The scheduler interprets a user's request and finds a suitable schedule for it by matching resource requirements with concrete resource descriptions, and trying to minimize the application completion time.
- **Restart on failure**: Both computation and communication jobs are automatically observed during the execution, and a re-scheduling policy can be implemented in the scheduler's Controller.
- **Extension**: The scheduler can load modules implementing different functionalities, each of which can be based on a different way of characterizing resources.

2.3.4 Involved resources

The extensibility of the KNOWLEDGE GRID scheduler allows the use of virtually any kind of resource needed by the users; the only limitations are those of resource providers.

2.3.5 Functional requirements

1. **Information retrieval**. The scheduler must be able to connect to external resource information services to retrieve data about (current and future) availability and performance of resources.
2. **Application parsing and validation**. The scheduler must parse and validate the scheduling requests with respect to their structure and with respect to the actual possibility to instantiate them.
3. **Resource pre-selection**. The available resources must be preliminarily filtered to retain only those actually usable for the application.
4. **Scheduling**. The scheduler must support a *scheduling process*, i.e. the sequence of actions to be taken in coincidence with particular events, and a *scheduling algorithm*, defining the way in which jobs are assigned to resources.
5. **Failure management**. The scheduling process must be *dynamic with re-scheduling*, i.e. the scheduler is invoked initially and then, during application executions, it is invoked again as a consequence of significant events occurred, to re-schedule unexecuted parts of the application.
6. **Extensibility**. It must be possible to extend the scheduler functionalities with personalized ones based on different application scenarios and Grid structures.

2.3.6 Workflow of scheduling process

For each application to be scheduled the scheduler instantiates a different Controller. Moreover, the following logical steps are performed (see also Figure 2.3-1):

1. The Cost/Size Estimator gathers data about characteristics and performances of available resources and builds the cost estimation functions (this step can be done offline).
2. The Matchmaker selects resources usable to execute the jobs composing the application, using information coming from the Resource Information Cache.
3. The Mapper evaluates a certain set of possible schedules, using information coming from the Estimation modules of the Cost/Size Estimator, and chooses the one minimizing the completion time.
4. The Controller requests job execution from the Execution Manager.
5. The Controller waits for job status notifications from the Job Monitor or new information about availability and performance of resources and adapts the schedule to such changes.

2.3.7 Involved scheduling components/services

The following service types are involved in the activity of the KNOWLEDGE GRID scheduler:

- **Data** and **Network Management** services.
- **Job Supervisor** service.
- **Information** service (static and forecasted).

2.3.8 Failure considerations

The KNOWLEDGE GRID scheduler handles job failures as described in Section 2.3.3 and Section 2.3.5.

2.3.9 Security considerations

Security in the KNOWLEDGE GRID scheduler is requested from other KNOWLEDGE GRID services; it is essentially based on GSI **Error! Reference source not found.**.

2.3.10 Accounting considerations

Accounting in the KNOWLEDGE GRID scheduler is requested from other KNOWLEDGE GRID services.

2.3.11 Performance considerations

The KNOWLEDGE GRID scheduler caches resource information and strongly indexes them in order to obtain the data access performance needed during the scheduling activity. In addition, due to the inherent intractability of the scheduling problem to be dealt with, one of the most important requirements of the scheduling heuristics is a suitable effectiveness/efficiency trade-off.

2.3.12 Use case situation analysis

We have designed a complete scheduling model and implemented the architecture described in Section 2.3.1 for its support. The scheduler is currently developed within the context of the KNOWLEDGE GRID, but its structure and openness show that it is suitable for more general scheduling scenarios. The study of suitable scheduling heuristics for different kinds of applications and Grids is currently underway.

2.4 GRASP (Grid Resource Allocation Services Package)

2.4.1 Summary

GRASP (Grid Resource Allocation Services Package) is designed to meet the requirements of the resource management problem dealing with delivering users plentiful computing power provided through distributed resources. Currently, the Managed Job Service of Globus Toolkit 3 is the service to be used to run a job on a remote resource. However, in order to build more useful Grids, there should be added some user-friendly resource allocation manners including resource brokering, scheduling, monitoring, and so forth in the collective layer. GRASP is aiming at this upper-GRAM level scheduling and job submission system. The following is a brief introduction of GRASP functions (see also Figure 2.4-1):

- **Grid job submission**: GRASP has a service, Job Submission Service, where users are interfacing with the Grid computing environment. The co-allocation problem has been solved for a cross-resource MPI-based parallel job by designing an MPICH initialization process in which all MPI sub-jobs are synchronized by the Job Submission Service. And also monitoring this service allows the user to monitor his/her job as a whole.
- **Resource brokering and meta-scheduling**: A Grid Scheduling Service finds resources fit to a user's job derived from a Grid information service. To select proper resources it performs matchmaking between a resource specification from the user and resource owner policies about jobs or users from each resource administrator. And then it selects resources to be allocated to the job from the candidates which have been found.
- **Local job execution**: A Resource Manager Service authenticates the user for the job execution on a local resource and submits the job to the local batch queuing system such as PBS. And this service will support the immediate reservation to minimize the failure of execution of a scheduled job in the upper layer during meta-scheduling.
- **Fault tolerant job execution**: Grids consist of so many computing resource components and each has a probability of local failure, which decreases the reliability of the whole Grid system. To increase the reliability of the system, fault tolerance for a Grid job is required. Without fault tolerance, parallel or distributed processes are vulnerable even at local single failure and might loose all computation mid-result on failure only to start from the beginning. In GRASP we realized a fault tolerant job execution which makes a Grid job to be restarted automatically from where the failure occurs, adopting the periodic check-pointing mechanism.

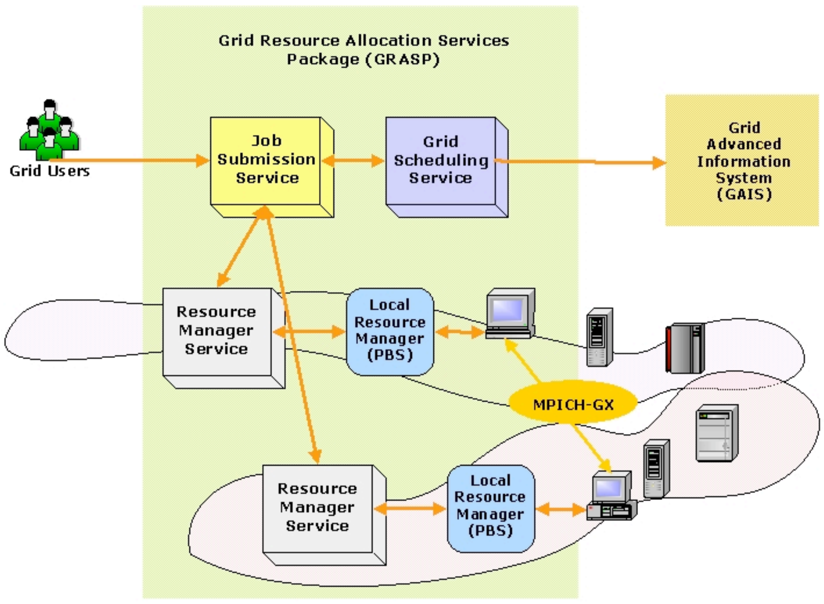
Grid Resource Allocation Services Package (GRASP)
Grid Users
Job Submission Service
Grid Scheduling Service
Grid Advanced Information System (GAIS)
Resource Manager Service
Local Resource Manager (PBS)
MPICH-GX
Resource Manager Service
Local Resource Manager (PBS)

Figure 2.4-1 Architecture of GRASP

2.4.2 Customers

The target customers of GRASP would be mostly computational scientists who used to run parallel jobs in a Grid environment.

2.4.3 Scenarios

- **Job submission**: Two major application types are considered: high throughput computing (HTC) and high performance computing (HPC) applications. In the case of high throughput computing, it is not necessary for each process on the resources to communicate with each other. Sensitivity analysis and parameter tuning studies are performed by high throughput computing methods. The other application is MPI-based parallel job for the high performance computing, which requires significant amount of communication among the sub-job processes. And also a hybrid of HTC and HPC, that is, a HTC job whose sub-jobs are MPI-based HPC jobs can be handled.
  In order to support these kinds of applications, GRASP interprets a user's job request, and then finds out and selects resources to appropriately run the job. After the scheduling process, the job is distributed to selected resources.
- **Job restart on failure**: In GRASP the MPI-based job can resume its computation automatically even when the job stops because of the failure of any sub-job process. There could be two kinds of failures on a distributed parallel job. One is a failure of a sub-job process running on a computing node, the other is a failure of a resource on which sub-job processes are running. When a process stops for its own reason, the cluster manager on the front node will fork a new process on that computing node (see also Figure 2.4-2). If the node is down, the cluster manager will choose another computing node in the cluster and resume the process on the node. More seriously if the whole cluster which the cluster manager is running on is down or the connection to the cluster is lost, the central manager in the Job Submission Service will choose another appropriate cluster and resume the sub-jobs on the cluster. As mentioned above, GRASP takes hierarchical failure recovery

system in which each failure manager handles the failures on each layer respectively.

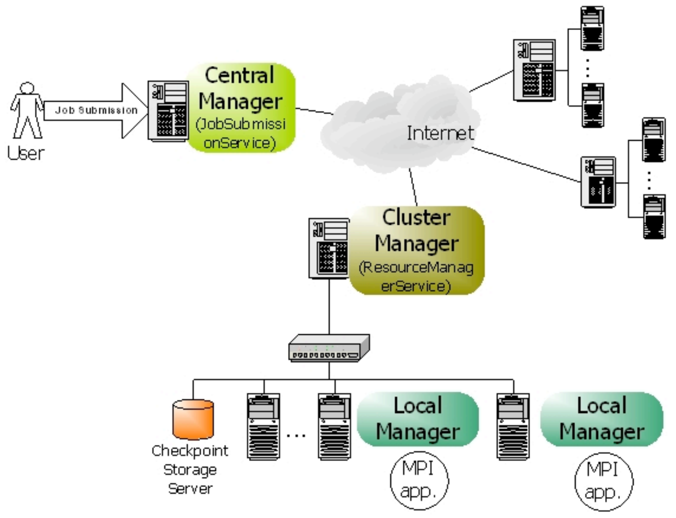
Job Submission
User
Central Manager (JobSubmissionService)
Internet
Cluster Manager (ResourceManagerService)
Checkpoint Storage Server
Local Manager
MPI app.
Local Manager
MPI app.

Figure 2.4-2 Fault tolerance job execution architecture

2.4.4 Involved resources

For now, GRASP is restricting resources to computing resources which are mainly clusters. However, the scope of resources will be gradually enlarged to storage devices, network connections, and so on.

2.4.5 Functional requirements

1. **Discovery and brokering**: For dynamic resource offering and user convenience, a Grid scheduling service should be able to discover and select proper resources from Grid environment. In this phase, the scheduling service would be aided by an information service.
2. **Queuing**: A Grid environment is so dynamic and unpredictable that a Grid job should wait in the queue until the scheduling process ends.
3. **Scheduling**: Scheduling is a process of matching a job to the appropriate resources. In this phase, various scheduling algorithm could be applied.
4. **Authentication and authorization**: Authentication and authorization is essential to run a job on a remote resource. Therefore the scheduling component should check if the user can acquire the admission to the resources.
5. **Advance reservation**: Although the advance reservation is required for Grid scheduling, it is not yet realized in GRASP. Therefore it is being worked on enabling the immediate reservation mechanism, in which the scheduling service can occupy the resources at the time when the meta-scheduling is done.
6. **Monitoring**: Monitoring job status and resource status could be considered. Job monitoring should be supported in the Grid scheduling components. Although resource monitoring is required to discover resources, it would be supported by an information service.
7. **Fault tolerance**: Fault tolerant job management can make a Grid system more effective because without a fault tolerance service, the computational results upon the job failure would be blown up and the job should start again from very first step.

2.4.6 Workflow of scheduling process

1. Queue the job for the scheduling.
2. Gather information about available resources from an information service.
3. Filter unsuitable resources using matchmaking between the job specification and the resource owner policies which should be offered by the information service.
4. Select the resources and number of nodes using various scheduling algorithm.
5. Reserve the resources based on the schedule.
6. Generate sub-job request scripts for each resource.
7. Submit each sub-job request to the resources.
8. Authenticate and authorize the user on the resources.
9. Verify the reservations.
10. Stage the required files on the resources.
11. Execute sub-jobs on the computing nodes.

2.4.7 Involved scheduling components/services

1. **Job submission service**: JSS (Job Submission Service) is responsible for management of the job. It receives a job request from clients, requests scheduling from a Grid scheduling service, initialize job execution through a local resource management services, and controls the jobs during execution with the job monitoring.
   GRASP supports an extended MPICH, which was implemented to make it possible for MPI sub-jobs that are dispersed on the remote resources to communicate with each other. In this mechanism, JSS plays an important role to synchronize sub-jobs by controlling the barriers in each sub-job process when an application is initialized.
   And also, JSS handles job failure as a central manager of fault tolerant job execution system. It synchronizes the check-pointing process on the resources with each other, and handles the failure on a resource level not on a computing node level.
2. **Grid scheduling service**: GSS (Grid Scheduling Service) discovers the resources available and chooses the best fitting resources for the job. In order to filter unacceptable resources, GSS does matchmaking between resource specification in the job request and resource owner's preference in the resource owner policy. The resource owner policy is delivered from the resource by an information service. Then the candidates selected from the matchmaking process enter the scheduling process and the final winners are picked out based on the scheduling algorithm. One example of the scheduling algorithm in GSS is point-based algorithm, in which all the resources have the point following user's preferences and the resources with higher points are selected. The last process in GSS is the reservation of the resources scheduled.
3. **Resource manager service**: RMS (Resource Manager Service) takes a job request from outside and starts the execution of the user program on the resource with some required functions, authentication, authorization, file staging, output and error streaming, local scheduler interfacing, and so forth. And also, during execution the job can be monitored and controlled by RMS. In addition to the basic function of job submission, RMS supports JSS in synchronizing MPI-based job and GSS in reserving the resource.

2.4.8 Failure considerations

GRASP can handle the job failure situations so that distributed processes do not lose their computation mid-results. The approach is to adopt the periodic check-pointing mechanism to decrease the loss of computation results. Check-pointing is an operation to store the state of a process into stable storage so that a process can resume its previous state at any time with the latest checkpoint file. In the described system (see also Figure 2.4-2), hierarchical job managers in JSS and RMS monitor and control MPI processes, that is to say, cluster manager in RMS and central manager in JSS are responsible for detecting node/network/process failures and deciding on a consistent global recovery strategy.

2.4.9 Security considerations

Security functionality of all services in GRASP is generally based on GSI **Error! Reference source not found.**. More precisely, RMS, the Grid service of a computing resource follows the authentication architecture of GT3 GRAM [19].

2.4.10 Accounting considerations

For accounting, the usage of computing resources should be measured correctly. This measuring is done by extracting information from the local scheduler on matching the local user account to the Grid user's credentials. The Grid user is represented by a distinguished name (DN) in the `gridmap` file.

2.4.11 Performance considerations

GSS applies a cache mechanism in order to make a good performance in fetching resource information. When discovering the resource information for a job, the local cache is searched for the resources satisfying the query at first. Only if the proper information could not be found in its local cache, GSS makes a query to an information service outside. The local cache is updated when new information reached from an information service, and updated by the Cache Auto Updater periodically using the notification mechanism in OGSI.

2.4.12 Use case situation analysis

GRASP is an ongoing development of an architecture to support scientific applications in a Grid environment. The implementation is not deployed in a real environment yet, but we are working on the deployment of GRASP in the Korean Grid infrastructure. The first targets would be applications from bio-informatics, computational fluid dynamics using genetic algorithms, and some data-intensive applications.

2.5 Scheduling in loosely-coupled Grids with GridWay

2.5.1 Summary

In spite of the great research effort made in Grid computing, application development and execution in the Grid continue requiring a high level of expertise due to its complex nature. In a Grid scenario, a sequential or parallel job is commonly submitted to a given resource by *manually* performing all the scheduling steps.

Moreover, one of the most challenging problems that the Grid computing community has to deal with is the fact that Grids are highly dynamic environments. An application should be able to adapt itself to rapidly changing resource conditions, namely: high fault rate and dynamic resource availability load and cost.

Therefore, in order to obtain a reasonable degree of both application performance and fault tolerance, a Grid scheduler must be able to adapt a given job according to the availability of the resources and the current performance provided by them. Grid*W*ay [20] is a Globus-based submission framework that allows an easier and more efficient execution of jobs on such dynamic Grid environments. Grid*W*ay automatically performs all the job scheduling steps, provides fault recovery mechanisms, and adapts job scheduling and execution to the changing Grid conditions.

2.5.2 Customers

This use case is intended for average Grid users, who mainly execute on the Grid compute-intensive stand-alone jobs with no special requirements, and high throughput computing applications. These jobs/tasks can be both parallel and sequential. There are two kinds of costumers, each one devoted to:

- **Execution**: The scheduler should provide an easy and efficient way to execute jobs on a Grid. The user specifies the Grid job through a job template, which contains all the necessary parameters for its execution in a *submit & forget* fashion. The underlying scheduling and execution system, automatically performs all the job scheduling steps, and watches for its correct and efficient execution.
- **Development**: Grid developers need an interface to distributed applications among Grid resources. These distributed applications consist mainly in communicating jobs that follow typical distributed paradigms like: asynchronous embarrassingly distributed, master worker, or complex workflows. The *Distributed Resource Management Application API* (DRMAA) [21] specification constitutes a homogenous interface to different Distributed Resource Management Systems (DRMS) to handle job submission, monitoring and control, and retrieval of finished job status. In this way, DRMAA could aid scientists and engineers to express their computational problems by providing a portable direct interface to DRMS.

2.5.3 Scenarios

We first describe the main characteristics and assumptions made about the scheduling scenario dealt by Grid*W*ay. There exist different kinds of Grids, from tightly-coupled environments, being dedicated to the execution of high-performance applications, to loosely-coupled systems, dedicated to the execution of high-throughput and complex applications. We focus on computational Grid infrastructures, build up from

uncoupled resources and interconnected by high-latency public networks, dedicated to the execution of high-throughput applications, which could be MPI-coded, and complex workflows, which consist of transformations performed on the data.
These Grid environments inherently present the following characteristics:

- Multiple administration domains and autonomy
- Heterogeneity
- Scalability
- Dynamism or adaptation

These characteristics completely determine the way that scheduling and execution on Grids have to be done. For example, scalability and multiple administration domains prevent the deployment of centralized resource brokers, with total control over client requests and resource status. On the other hand, the dynamic resource characteristics in terms of availability, capacity and cost, make essential the ability to adapt job scheduling and execution to these conditions. Finally, the management of resource heterogeneity implies a higher degree of complexity.
From the user point of view, we assume the following application model:

- **Executable file:** The executable must be compiled for the remote host architecture. The scheduler should provide a straightforward method to select the appropriate executable for each host.
- **Input files:** These files are staged to the remote host. The scheduler should provide a flexible way to specify input files and supports Parameter Sweep like definitions. Please note that these files may be also architecture dependent.
- **Output files:** These files are generated on the remote host and transferred back to the client once the job has finished.
- **Standard I/O streams:** The standard input file is transferred to the remote system previous to job execution. Standard output and standard error streams are also available at the client once the job has finished.
- **Re-start files:** Restart files are highly advisable if dynamic scheduling is performed. User-level check-pointing managed by the programmer must be implemented because system-level check-pointing is not possible among heterogeneous resources.

In addition the user must specify a set of job requirements, and a ranking criterion (see Section 2.5.7 below). Also, it is possible for the application to generate a performance profile as a registry of its activity (see Section 2.5.11 below).

2.5.4 Involved resources

This use case focuses on computational resources. In the present context we adopt a rather simple management of storage resources, but more advanced techniques (like third-party transfers, meta-data catalogues, access to online databases…) could be possible.

2.5.5 Functional requirements

1. **Support for adaptive scheduling**: Given the dynamic characteristics of Grid environments, it is necessary to periodically re-evaluate the schedule initially performed. So, the schedule can be dynamically adapted to the available resources and their characteristics, normally considering the number of pending and running jobs, and the history profile of completed jobs. *Adaptive scheduling* has been widely studied in the literature, and it has been

demonstrated that periodic re-evaluation of the schedule can result in significant improvements in both performance and fault tolerance.

2. **Support for adaptive execution**: Additionally, it could be necessary to migrate running applications to more suitable resources. *Adaptive execution* can improve application performance by adapting it to the dynamic availability, capacity and cost of Grid resources. In this case the overhead induced by job migration is the key issue that must be considered. Migration is commonly implemented by restarting the job on the new candidate host. Therefore, the job should generate restart files at regular intervals in order to restart execution from a given point. However, for some application domains the cost of generating and transferring restart files could be greater than the saving in compute time due to check-pointing. Hence, if the check-pointing files are not provided the job is restarted from the beginning. In order not to reduce the number of candidate hosts where a job can migrate, the restart files should be architecture independent.
3. **Support for self-adaptive applications**: An application could take decisions about resource selection as its execution evolves, and provide its own performance activity to detect performance slowdown.
4. **Fault tolerance**: Job failures should be automatically detected, allowing the user to abort or retry it execution or automatically migrating it to a new machine.
5. **Log information**: Most relevant information about the jobs should be obtained from several log files.
6. **Unix-like command interface**: Commands should be very similar to those found on UNIX systems and resource management systems like PBS or SGE. Users should be able to submit, kill, migrate, watch and wait for jobs or array of jobs.
7. **Programming interface**: DRMAA should be supported to develop distributed applications.
8. **Use of standard Grid services**: It should be based on the functionality provided by Globus basic services [22]:
   - Grid security infrastructure: GSI
   - Grid execution service: GRAM
   - Grid information service: MDS
   - Grid file transfer service: GridFTP
9. **Extensibility and adaptability of the functionality**: the architecture should be modular in order to support different middleware versions (*middleware access driver* and *wrapper* modules for access Grid execution services, *resource selector* module for access Grid information services, *prolog*/*epilog* modules for access Grid file transfer services). Moreover, the architecture should be decentralized, allowing to be deployed as a client tool.

2.5.6 Workflow of scheduling process

In a Grid scenario, a sequential or parallel job is commonly submitted to a given resource by taking the following path [23]:

1. **Resource discovery and selection**: Based on a set of job requirements, like operating system or platform architecture, a list of appropriate resources is obtained by accessing to an information service mechanism. Then a single resource is selected among the candidate resources in the list.

2. **Preparation**: The selected host is prepared for job execution. This step usually requires staging of input files.
3. **Submission and migration**: The job is submitted to the selected resource. However, the user may decide to restart its job on a different resource, if a performance slowdown is detected or a *better* resource is discovered.
4. **Monitoring**: The job evolution is monitored over time.
5. **Termination**: When the job is finished, its owner is notified and some completion tasks, such as output file staging and cleanup, are performed.

2.5.7 Involved scheduling components/services

The core of the Grid*W*ay framework [24] is a personal submission agent that performs all submission stages and watches over the efficient execution of the job. Adaptation to changing conditions is achieved by dynamic rescheduling. Once the job is initially allocated, it is rescheduled when performance slowdown or remote failure are detected, and periodically at each *discovering* interval. Application performance is evaluated periodically at each *monitoring interval* by executing a *performance evaluator* program and by evaluating its accumulated *suspension time*. A *resource selector* module acts as a personal resource broker to build a sorted list of candidate resources.
The submission agent consists of the following components (see Figure 2.5-1):

- **Request manager** (RM): To handle client requests.
- **Dispatch manager** (DM): To perform job scheduling.
- **Submission manager** (SM): To execute and migrate jobs and monitor its correct execution.
- **Performance monitor** (PM): To evaluate the job performance.

The flexibility of the framework is guaranteed by a well-defined API (Application Program Interface) for each submission agent component. Moreover, the framework has been designed to be modular to allow adaptability, extensibility and improvement of its capabilities. The following modules can be set on a per job basis:

- **Resource selector** (RS): Used by the *dispatch manager* to select the most adequate host to run the job according to the host’s rank, architecture and other parameters.
- **Middleware access driver** (MAD): Used by the *submission manager* to provide an interface with the underlying resource management middleware.
- **Performance evaluator** (PE): Used by the *performance monitor* to check the progress of the job (not supported in the current version).
- **Prolog**: Submitted by the *submission manager* to create the job directory hierarchy on the remote machine and transfer the executable, input and restart files.
- **Wrapper**: Submitted by the *submission manager* to run the executable file and captures its exit code.
- **Epilog**: Submitted by the *submission manager* to transfer output or restart files, clean up the GASS cache and remove the job directory from the remote machine.

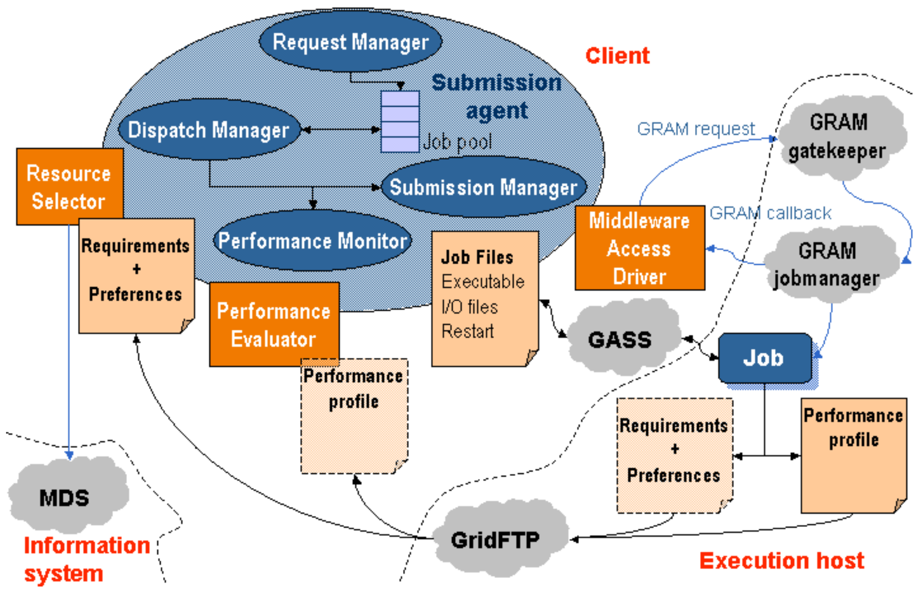

Figure 2.5-1 GridWay services

The following actions are performed by the submission agent:

- The client application uses a Client API to communicate with the *request manager* in order to submit the job along with its configuration file, or job template, which contains all the necessary parameters for its execution. Once submitted, the client may also request control operations to the *request manager*, such as job *stop*/*resume*, *kill* or *reschedule*.
- The *dispatch manager* periodically wakes up at each *scheduling* interval, and tries to submit *pending* and *rescheduled* jobs to Grid resources. It invokes the execution of the *resource selector* module corresponding to each job, which returns a sorted list of candidate hosts. The *dispatch manager* submits *pending* jobs by invoking a *submission manager*, and also decides if the migration of *rescheduled* jobs is worthwhile or not. If this is the case, the *dispatch manager* triggers a *migration* event along with the new selected resource to the job *submission manager*, which manages the job migration.
- The *submission manager* is responsible for the execution of the job during its lifetime, i.e. until it is *done* or *stopped*. It is initially invoked by the *dispatch manager* along with the first selected host, and is also responsible for performing job migration to a new resource. The Globus management components and protocols are used to support all these actions through the *middleware access driver*. The *submission manager* performs the following tasks (see Figure 2.5-2):
  - **Prologing**: Preparing the RSL (Resource Specification Language [25]) job description and submitting the *prolog* executable.
  - **Submitting**: Preparing the RSL, submitting the *wrapper* executable, monitoring its correct execution (as explained in subsequent sections), updating the submission states via GRAM call-backs and waiting for *migration*, *stop* or *kill* events from the *dispatch manager*.
  - **Cancelling**: Cancelling the submitted job if a *migration*, *stop* or *kill* event is received by the *submission manager*.
  - **Epiloging**: Preparing the RSL and submitting the *epilog* executable.
- The *performance monitor* periodically wakes up at each *monitoring* interval. It requests *rescheduling* actions to detect better resources when performance slowdown is detected and at each *discovering* interval.

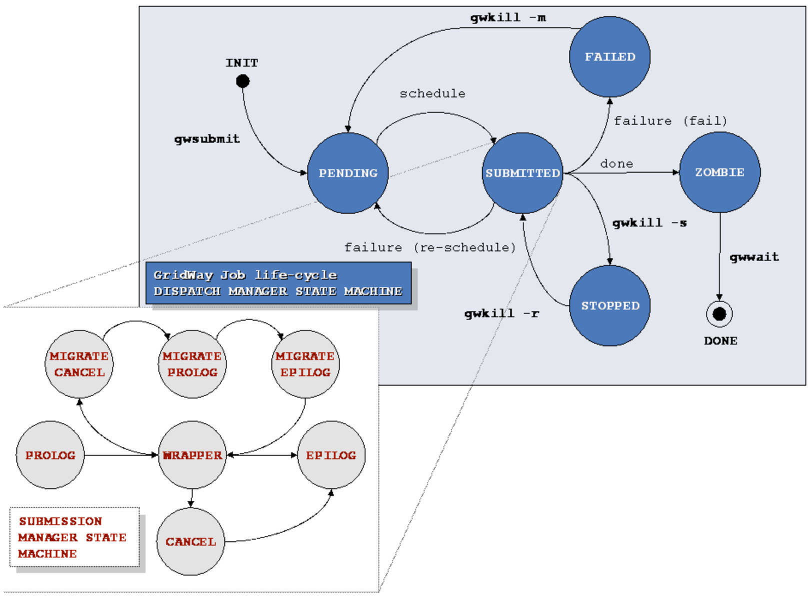

Figure 2.5-2 Submission manager state machine & job life-cycle

Due to the heterogeneous and dynamic nature of the Grid, the end-user must establish the requirements that must be met by the target resources (discovery process) and criteria to rank the matched resources (selection process). The attributes needed for resource discovery and selection must be collected from the information services in the Grid testbed, typically the Globus Monitoring and Discovery Service (MDS). Usually, resource discovery is only based on static attributes (operating system, architecture, memory size...) collected from the Grid Information Index Service (GIIS), while resource selection is based on dynamic attributes (disk space, processor load, free memory...) that can be obtained from the Grid Resource Information Service (GRIS) or by accessing the Network Weather Service.

The *resource selector* is executed by the *dispatch manager* in order to get a ranked list of candidate hosts when the job is pending to be submitted or a rescheduling action has been requested. The *resource selector* is a script or a binary executable, specified in the job template, which receives the parsed job template itself as an argument (so it can be easily sourced in a script) and some other needed parameters in the environment.

The *resource selector* module performs the following actions (see Figure 2.5-3):

1. Available compute resources are discovered by accessing the GIIS server and, those resources that do not meet the user-provided host requirements are filtered out. At this step, an authorization test (via a GRAM ping request) is also performed on each discovered host to guarantee user access to the remote resource.
2. Then, the dynamic attributes of each host and the available GRAM *job managers* are gathered from its local GRIS server.
3. This information is used by an user-provided rank expression to assign a rank to each candidate resource.

Finally, the resultant prioritized list of candidate resources is used to dispatch the job.

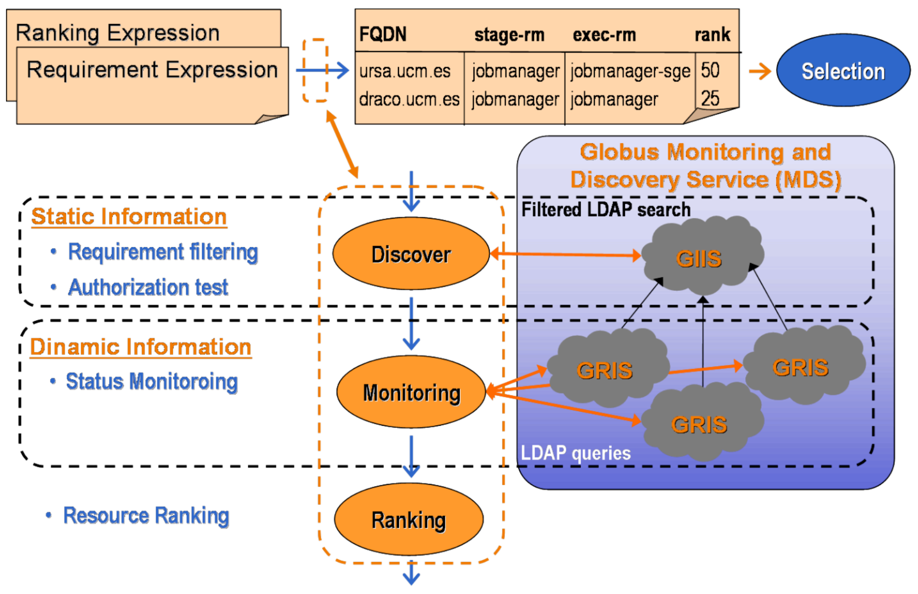

Figure 2.5-3 Resource selection

Job execution is performed in three steps by the following modules:

- **Prolog**: This module is responsible for creating the remote experiment directory and transferring the executable and all the files needed for remote execution, such as input or restart files corresponding to the execution architecture. These files can be specified as local files in the experiment directory or as remote files stored in a file server through a GridFTP URL. Once the files are transferred to the remote host, they are added to the GASS cache so they can be re-used if they are shared with other jobs.
- **Wrapper**: This module executes the submitted job and writes its exit code to standard output, so the submission agent can read it and can be used to determine whether the job was successfully executed or not. The capture of the remote execution exit code allows users to define complex jobs, where each depends on the output and exit code from the previous job. It is interesting to note that versions previous to 3.9 of the Globus Toolkit do not provide any mechanism to capture the exit code of a job.
- **Epilog**: This module is responsible for transferring back output files, and cleaning up the remote experiment directory. At this point, the files are also removed from the GASS cache.

File transfers are performed through a reverse-server model. The file server (GASS or GridFTP) is started on the local system, and the transfer is initiated on the remote system using Globus transfer tools (i.e. `globus-url-copy` command).

The *prolog* and *epilog* modules are always submitted to the fork GRAM *job manager*. In this way, our tool is well suited for closed systems, such as clusters, where only the front-end node is connected to the Internet and the computing nodes are connected to a system area network, so they are not accessible from the client.

Job migration is performed in the following way. The execution of the *wrapper* module is cancelled (if it is still running). Then, the *prolog* module is submitted to the new candidate resource, preparing it and transferring all the needed files to it, including the restart files from the old resource. After that, the *epilog* module is submitted to the old resource (if it is still available), but no output file staging is performed, it only cleans up the remote system. And finally, the *wrapper* module is submitted to the new candidate resource.

Briefly, the execution of jobs in three separate steps has the following advantages:

- Support for closed systems (like clusters).
- Easy and efficient way to implement job migration.
- Possibility to separately schedule transfers and executions.
- Better adjustment of job definition parameters (expressed using the RSL language), as `MaxCPUTime`.
- Implementation of different transfer strategies (caching, compression, access to replica catalogues or online databases…) does not affect the compute nodes.

2.5.8 Failure considerations

Grid*W*ay provides the application with the fault detection capabilities needed in such a faulty environment:

- The GRAM *job manager* notifies submission failures as GRAM call-backs. This kind of failures includes connection, authentication, authorization, RSL parsing, executable or input staging, credential expiration and other failures.
- The GRAM *job manager* is probed periodically at each *polling* interval.
- The standard output of prologue, wrapper and epilogue is parsed in order to detect failures. In the case of the wrapper, this is useful to capture the job exit code, which is used to determine whether the job was successfully executed or not. If the job exit code is not set, the job was prematurely terminated, so it failed or was intentionally cancelled.

When an unrecoverable failure is detected, Grid*W*ay retries the submission of prologue, wrapper or epilogue a number of times specified by the user and, when no more retries are left, it performs an action chosen by the user among two possibilities: stop the job to manually resuming it later, or automatically generate a rescheduling event.

2.5.9 Security considerations

In a loosely coupled environment, security decisions must be performed at the resource or site level.

2.5.10 Accounting considerations

The `gwps` and `gwhistory` commands provide status and accounting information about the submitted hosts. Nevertheless, in a loosely coupled environment, accounting must be performed at the resource or site level.

2.5.11 Performance considerations

The framework provides two mechanisms to detect performance slowdown:

- A *performance evaluator* is periodically executed at each monitoring interval by the *performance monitor* to evaluate a rescheduling condition. Different strategies could be implemented, from the simplest one based on querying the Grid information system about workload parameters to more advanced strategies based on detection of performance contract violations. The *performance evaluator* is a script or a binary executable specified in the job template, which can also include additional parameters needed for the performance evaluation. A mechanism to deal with application own metrics is provided since the files processed by the *performance evaluator* could be dynamically generated by the running job. The rescheduling condition verified by the *performance evaluator* could be based on the performance history using advanced methods like fuzzy logic, or comparing the performance with the initial performance attained, or a base performance.

- A running job could be temporally suspended by the resource administrator or by the local queue scheduler on the remote resource. The submission agent takes count of the overall *suspension time* of its job and requests a *rescheduling* action if it exceeds a given threshold. Notice that the *maximum suspension time* threshold is only effective on queue-based resource managers.

2.5.12 Use case situation analysis

Grid*W*ay is currently available [20] with some limitations.

2.6 Agent-based scheduling with Calana

2.6.1 Summary

The concept of virtual organizations has a big impact on resource usage: Grid computing enables organizations to use the resources of different organizations seamlessly. Beside the benefits, this usage pattern also raises new problems: Organizations want to enforce their policies when advertising their resources. When using resources of other organizations, users may want to specify their goals on a per-job basis.
This use case describes a market-based attempt to enable all stakeholders in a Grid to integrate their individual policies in the resource allocation process. As the real economy shows, markets are a powerful mean to coordinate many market participants.

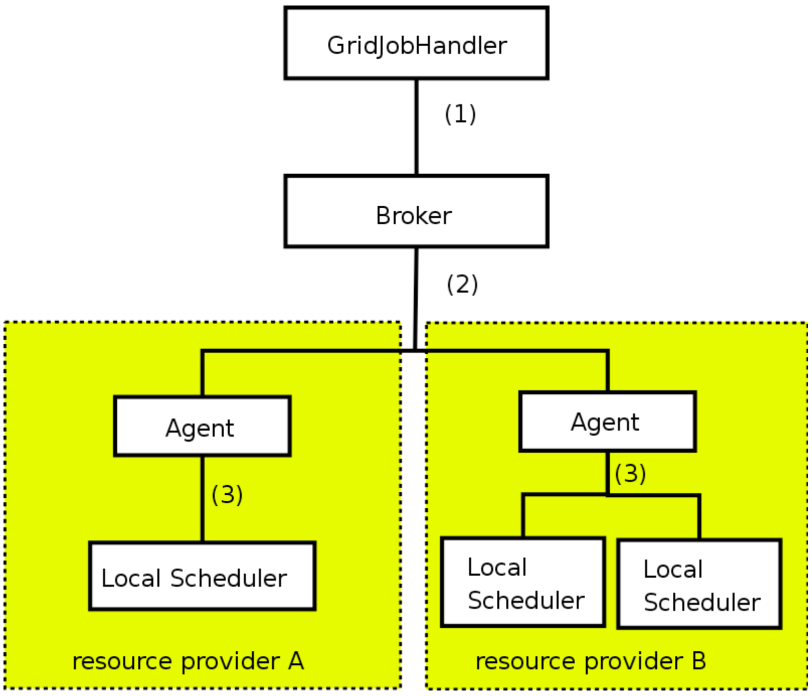

Figure 2.6-1 Architecture overview: The GridJobHandler uses the broker component to create a valid schedule (1). Resource providers use an agent in order to connect their local scheduling systems to the architecture (3). The broker and agents interact to create the schedules (2).

Fraunhofer ITWM's Calana [26] implements a market-based scheduler. As section 2.6.6 shows, Calana uses auctions to allocate resources. It consists of two components: The broker and the agent. Users sent the job scheduling request along with some preference information to the broker component, which then starts an auction by notifying all attached agents. The agents are running at the provider's site and send bids to a central broker in order to run a job. They can implement various bidding strategies in order to meet their local policies. When the auction finishes, all bids are compared to each other. The user's preferences are taken into account by judging the bids accordingly. The best bid wins the auction and the respective resource will be the execution location for the job.

As an additional benefit, there is no need for a centralized information system to provide dynamic information: Since the agents reside at the provider's site, they can retrieve all necessary information locally.

2.6.2 Customers

Calana is designed as a general-purpose Grid scheduler and therefore aims at no specific user group. However, based on the architecture of Calana, it is assumed that there will be three stakeholder groups:

1. Providers of computational power may use an agent-based scheduler to sell their computational power as well as any other resource, e.g. software licenses.
2. Users use Calana to find cheaper or more powerful resources.
3. A computational economy allows service providers to act as a merchant without owning resources: for example, a merchant may bid on a job issued by one broker and start a second auction at another broker to obtain the resources needed to execute the job.

2.6.3 Scenarios

In general, there are no limitations on the usage of Calana. Since each resource provider can adjust the agent to interact with the local resource management system, almost all resources can be integrated.
Different from other use cases, the strategy of all stakeholders is expected to be more important.

2.6.4 Involved resources

Since Calana relies on GADL [27], there is no limitation regarding the type of resource. The GADL provides a XML-based language to describe resources and workflows. Each resource needs to provide a resource description for the static properties. The dynamic properties are handled by the agent.

2.6.5 Functional requirements

1. **Authentication, authorization, user right delegation & job integrity verification**. Authentication and authorization are essential for every Grid based job submission scenario. To enable the scheduler to act on behalf of the user, the respective rights have to be delegated from the user to the scheduler. This use case also requires that the integrity of a job (parts of the job) can be verified anytime during the scheduling process.
2. **Job parsing & validation.** The job description has to be parsed and formally validated (job pre-processing).
3. **Information retrieval (static & dynamic)**. The dynamic information will be retrieved locally during the scheduling (see below). Calana does not rely on a centralized information system for dynamic information. Static information is needed to find execution candidates. This is currently done by using the GADL.
4. **Resource pre-selection.** This is done implicitly within Calana: By submitting a bid, an agent states that the corresponding provider has a resource the job will be able to run on. The agent can decide this by retrieving the GADL descriptions.
5. **Scheduling**. The scheduling decision is made by an auction, see Section 2.6.6.
6. **Advance reservation**. Calana relies on an advance reservation for each provider in order to ensure the job can be executed at the provider's site.

7. **Workflow execution/processing**. Calana relies on the GridJobHandler [28] in order to execute workflows. The GridJobHandler is capable of executing workflows specified in GADL.
8. **Billing/accounting** Since all scheduling decisions are made by the broker, it is possible to track prices etc. here.
9. **Failure management**. The GridJobHandler reacts on failures. This way, it is possible to specify alternative workflows in case of failures. The scheduler does not implement special techniques.

2.6.6 Workflow of scheduling process

The scheduling workflow corresponds to an auction. These steps can distinguish:

1. The request is received.
2. A “Call for Bids” is sent to all attached agents, containing information about the job and the estimated runtime of the job (e.g., a normalized wall time) along with information about the user.
3. The provider’s agent considers the creation of a bid. It has to judge whether the job can be executed at the local site or not. The GADL description contains all needed static information. If the local strategy suggests bid creation, the agent obtains an advance reservation from the target machine and submits a bid to the broker.
4. The broker then judges all bids, taking the user’s preferences into account.
5. The agents are notified about the winner of the auction.
6. The schedule is delivered.

The result is a schedule which can be used by the GridJobHandler to execute the job.

2.6.7 Involved scheduling components/services

Involved components are:

1. The GridJobHandler.
2. The Calana Broker.
3. Various Calana agents.
4. Local resource management systems.

2.6.8 Failure considerations

Based on section 2.6.6, the following failures have to be taken into account:

- *(Processing of the job request)*
  1. The parser doesn’t support the job description
- *(Gathering of static information)*
  2. The included GADL references can not be resolved.
- *(Scheduling)*
  3. No suitable resource found.
  4. Timeout.
  5. Local resource information cannot be retrieved.
- *(Advance reservation)*
  6. No advance reservation can be made
- *(Workflow execution/processing)*
  7. Any error during the execution of the workflow is handled by the GridJobHandler.

2.6.9 Security considerations

The security considerations do not differ significantly from the other use cases.

2.6.10 Accounting considerations

As stated above, the broker can also serve as an accounting service: simple logging of the auction data provides all basic data needed for accounting purposes on the inter-domain level. For the local domain accounting, each provider's agent may log the auction it has won.

2.6.11 Performance considerations

As in use case "Scheduling complex workflows" (see Section 2.1), the communication seems to have the main impact. Since the auction announcements are broadcasted, they are the limiting factor.

1. **Service programming model.** A multi-agent system is used to implement the architecture. Both broker and provider agents can be seen as individual agents, connected by a message layer that provides secure message transport. Currently, XMPP is used as messaging protocol [29].
2. **Communication failure.** XMPP takes care of all message handling issues.
3. **Scalability.** The number of messages directly corresponds to the number of attached agents. However, since the message transfer can be handled by separate XMPP servers that may also be distributed, it is assumed that the traffic can be handled. Furthermore, one may also choose to implement a hierarchy of brokers in order to partition the attached agents.
   In addition, since Calana doesn't rely on a centralized information system for dynamic information, the additional periodic updates of the information system doesn't occur.

2.6.12 Use case situation analysis

Calana is currently in the beta stage. It will be improved continuously and available under terms of the GPL. Integration with various resource management systems, e.g. Globus and UNICORE, will be available. It is planned to use this solution in various industry projects.

2.7 Meta-scheduling/co-allocation in the VIOLA testbed

2.7.1 Summary

The VIOLA (Vertically Integrated Optical Testbed for Large Applications) project aims to use novel network techniques and innovative services in an optical testbed. These techniques and services will be integrated and evaluated with challenging applications [30].
The applications selected are parallel and distributed simulations, like multi-physics simulations for example. Here different parts of the simulation are performed on different computational resources connected through an optical network. The user specifies the requirements of the simulation job using the UNICORE Client [31]. In order to run such a simulation both the necessary compute resources and the interconnecting network with a defined QoS (e.g. minimal bandwidth and delay) must be available at execution time. The necessary co-allocation is organised by a MetaScheduling Service [32] which is part of the testbed middleware, interacting both with UNICORE (the Grid middleware of the testbed) and with all local resource management and scheduling systems (RMS). The MetaScheduling Service is responsible for negotiating a common time-slot with all local RMS where all required resources (e.g. PC clusters, network links with a given QoS, visualisation devices) are available to run the simulation. Once such a common time-slot is identified the resources are reserved at all sites for this time-slot. The resulting agreement on the reservation [33] is then passed to the UNICORE client which transfers the job to the UNICORE system where the individual parts of the simulation are managed and monitored in the usual way.

Therefore, the main objectives of the MetaScheduling Service are:

- *Automation of co-allocation of computational and network resources.* With the use of the UNICORE Client, users are allowed to specify the resource requirements of their application while the low-level scheduling aspects of the co-allocation are managed transparently by the service.
- *Support for complex distributed applications*. Given a set of available heterogeneous clusters with different local application environments, the service allows to use several distinct clusters in parallel. Thus distributed applications with a need for particular hardware and software to execute their components may be run effectively.
- *Application performance improvement*. Given a set of available clusters, the service allows to use several clusters in parallel for one application thus trying to minimize applications' completion times.
- Extend the UNICORE Web Service Agreement based resource management framework with a co-allocation service.

Besides the MetaScheduling Service itself adapters for the different local RMS are part of the environment. These adapters are implemented as Web Services and may thus be operated somewhere in the network, either local to the RMS or at the site hosting the MetaScheduling Service.

The architecture of the UNICORE MetaScheduling environment comprises five main components (Figure 2.7-1):

- *UNICORE Client*. The graphical user interface to the UNICORE system. The user specifies his job together with the resources needed to run the job. The client also does a certificate-based authorisation of the user.
- *MetaScheduling Service*. Does the negotiations with the local RMS to identify a common time-slot to run the user's job based on the requirements received from the UNICORE Client.
- *Adapter*. Hides the specifics of the local RMS from the MetaScheduling Service and allows the service to use a single interface to all RMS.
- *Local RMS*. Does the local scheduling of the submitted job components based on the reserved time-slot negotiated with the MetaScheduling Service.
- *UNICORE system*. The Grid middleware that is itself composed of three major components: the UNICORE Gateway, the Network Job Supervisor (NJS), and the Target System Interface (TSI). The NJS receives the job description via the Gateway from the UNICORE Client. The system manages and monitors the user's job across multiple sites.

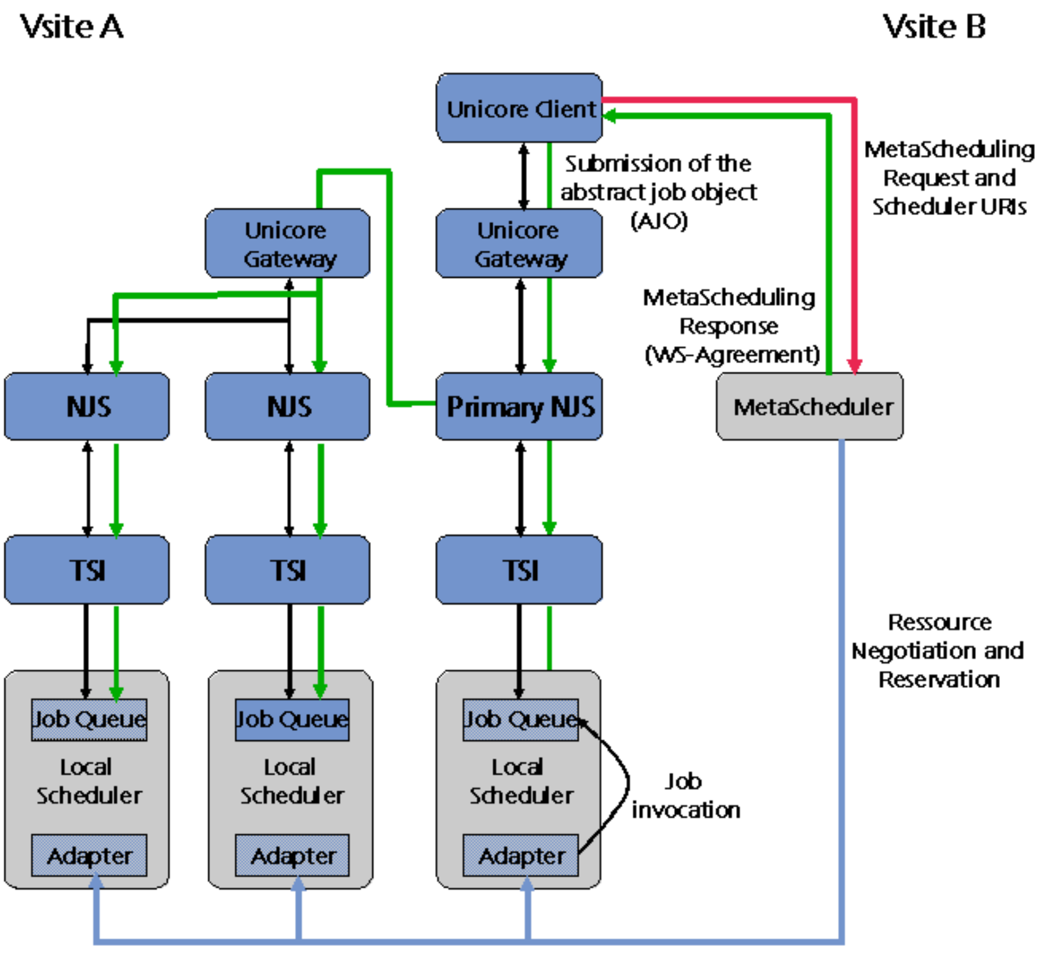

Figure 2.7-1 VIOLA meta-scheduling architecture

The MetaScheduling Service provides an open interface based on WS-Agreement allowing to be used in any Web Service compliant environment.

2.7.2 Customers

The target customers of the MetaScheduling Service are Grid users with a need to use more than one Grid resource exclusively for their application at the same time, for example two clusters and the interconnecting network with a dedicated QoS. The current implementation of the service is on top of the UNICORE middleware. However, since the MetaScheduling Service is not tightly coupled with the UNICORE system, it may easily be used with other Grid middleware like Globus-based Grids.

2.7.3 Scenarios

- **Job/Application submission**: The scheduler interprets a user's request and finds a suitable schedule by negotiating a common time-slot. Once the time-slot is identified the resources are reserved and the information is passed back to the UNICORE Client.
- **Job/Application cancelling**: If the job is cancelled using either the UNICORE Client interface or the MetaScheduling Service API all sub-jobs are aborted and all resources allocated are freed.
- **Free resources on failure**: If one of the sub-jobs fails during execution the whole job will be cancelled and all resources allocated to the job will become available again for the local RMS.

2.7.4 Involved resources

The use of an adapter as the interface between the MetaScheduling Service and the local RMS allows the co-allocation of virtually any kind of resource needed by the users; the only limitations are those imposed on individual RMS which must

- be capable of publishing at least the next possible start time for a given job and
- support advance reservation of a fixed time-slot.

2.7.5 Functional requirements

1. **Information retrieval**: The MetaScheduling Service must be able to connect to the remote RMS to retrieve data about (current and future) availability of resources and QoS.
2. **Negotiation of a common time-slot**: The MetaScheduling Service must parse and validate the possible next starting times of the remote RMS until a common time-slot has been identified or the end of the scheduling preview period of one of the systems has been reached.
3. **Resource reservation**: The required resources must be reserved for later use during the common time-slot.
4. **WS-Agreement**: The MetaScheduling Service must support the WS-Agreement protocol for the communication with the external (graphical) user interface (the UNICORE Client in case of the VIOLA Grid testbed).
5. **Failure management**: The MetaScheduling Service must dynamically connect to the remote RMS if the job (partly) fails to signal this failure and request freeing of the resources.

2.7.6 Workflow of scheduling process

For each meta-application to be scheduled, the UNICORE Client, the MetaScheduling Service and the UNICORE middleware perform the following logical steps:

1. After the user defined the description of the job and the resource requirement, the UNICORE Client (in VIOLA - or any other appropriate Grid middleware

front-end in the case of another Grid middleware) sends an “Agreement Offer” [33] specifying the resources requested by the user and the user’s IDs at the remote systems to the MetaScheduling Service

2. The MetaScheduling Service starts the negotiation with the remote RMS to identify a common time-slot to run the job.
3. Once the negotiation completes successfully the MetaScheduling Service does the reservation of all requested resources for the identified common time-slot.
4. The MetaScheduling Service sends the offer response back, i.e. the agreement, back to the client.
5. The Client creates the UNICORE Abstract Job Object (AJO) and sends this through the UNICORE gateway to the UNICORE Network Job Supervisor.
6. The NJS analyses and processes the AJO, enables monitoring and tracking of the job, and hands the job-date over to the UNICORE Target System Interface.
7. The TSI finally does the mapping from the AJO to a concrete job for the local system and passes the job to the local system where the reservation made in 4, is used for the execution of the job.

2.7.7 Involved scheduling components/services

The following services are involved in the activity of the UNICORE MetaScheduling Service:

1. Compute Resource Management services (local schedulers)
2. Network Management service (dedicated RMS for network resources)
3. WS-Agreement service
4. Job Supervisor service
5. Information service

2.7.8 Failure considerations

The MetaScheduling Service handles job failures as described in Section 2.7.3.

2.7.9 Security considerations

Security in UNICORE MetaScheduling framework is addressed on several levels:

- The user authenticates himself with his personal X.509 certificate managed through the UNICORE Client (single sign-on).
- The X.509 certificate is further used to retrieve the local IDs of the certificate owner at the different sites.
- The MetaScheduling Service does reservations of the requested resources on behalf of the user with his respective IDs at the local sites.
- The communication between the MetaScheduling Service and the Adapters are mutually secured with client/server certificates of the involved hosts.

2.7.10 Accounting considerations

Accounting of the co-allocated resources is done by the local systems (as well billing, if any). Currently there is no consolidation of the distributed accounting data, but this could be easily provided by either the MetaScheduling Service or the UNICORE system if the local systems provide an interface to access these data.

2.7.11 Performance considerations

The MetaScheduling Service uses an effective algorithm do determine a common time-slot for an application using several resources. The current implementation of this algorithm assumes that the local RMS will not expose the entire schedule, but provide information about the next possible start-time for the request only. The performance of the algorithm could be improved in a "friendly" environment where the local systems publish a larger part of their schedule. Thus the performance of the algorithm is to some extend dependent on the behaviour of the remote systems.

2.7.12 Use case situation analysis

The MetaScheduling Service has been designed and implemented within the architecture described in Section 2.7.1 (the VIOLA testbed). The MetaScheduling Service currently is used in the UNICORE based Grid of the VIOLA testbed, but its structure and openness are suitable to be integrated into other middleware environments.

3 Usage patterns

The purpose of this section is to describe common patterns which one can determine when dealing with Grid scheduling and resource management. The list of patterns is neither exhaustive nor does it provide a normative terminology of Grid scheduling and resource management usage scenarios. But the patterns together with the use cases are the basis for defining Grid scheduling requirements and specifying the services needed to fulfil these requirements.
For each pattern the scenario is briefly outlined and the characteristics concerning Grid scheduling–specific properties are listed. Please note that it is impossible (and not necessary) to clearly define boundaries between the different usage patterns. They primarily help readers to roughly classify their usage scenario and to get an idea of the implications on the functions needed.

3.1 Simple job submission

Scenario
The user or agent acting on behalf of the user is submitting a simple job to a Grid scheduler. The scheduler decides, based on information it is able to receive from the local resource management systems/schedulers, which resource is suitable to execute the job. Possible criteria the scheduler may evaluate are resource availability, the current load of the resource, queue lengths or the response time of previous requests. The scheduler then forwards the job to the local resource manager without further processing.
This profile is not Grid-specific and covers some HPC scenarios where resources are controlled by queuing systems. Apart from the "Grid" scheduler no specific Grid middleware is required.

Characteristics:

- The Grid scheduler needs information about the local resources, a demand which neither implies a specific information service (although helpful) nor does the existence of a Grid scheduler requires specific information to be delivered from the local resource managers/schedulers (although helpful).
- No complex workflows have to be managed, except if they are processed by the local resource manager/scheduler.
- No advance reservation is needed.
- No co-allocation is needed.
- Lower-level schedulers do not have to support any mechanisms to impose/negotiate service level guarantees.

3.2 Single-site execution with service guarantees

Scenario:
This scenario is similar to the Simple Job Submission pattern (see Section 3.1), except that the local resource management system must be capable of providing information about service guarantees. A typical example for such a pattern is a job being submitted to a local resource management system which uses a queuing mechanism combined with a backfilling strategy. Such a system can provide information about the latest end time of a job.

Characteristics:

- No complex workflows have to be managed, except if they are processed by the local resource manager/scheduler.
- No advance reservation is needed.
- No co-allocation is needed.
- Lower-level schedulers provide information about service guarantees (e.g. the start and end time of the submitted job).

3.3 Job execution with advance reservation

Scenario:
In addition to the usage pattern in Section 3.2 this one requires local management systems which include means to reserve a resource for a distinct period of time and guarantee that the user/agent can allocate this resource for the requested time.

Characteristics:

- No complex workflows have to be managed, except if they are processed by the local resource manager/scheduler.
- The local resource management system has to have advance reservation capabilities.
- No co-allocation is needed.
- Although this usage pattern does not cover the co-allocation of resources, advance reservation is a crucial prerequisite for doing co-allocation.

3.4 Co-allocation

Scenario:
In addition to the advance reservation capability needed on the local resource management level (as outlined in Section 3.3), co-allocation requires further functions on the Grid scheduling level. In this scenario information is needed about the availability of resources and their respective level of service to negotiate and plan a synchronised allocation of a set of resources. Co-allocation in this case refers to the parallel allocation of all resources involved, like it is e.g. required to execute MPI jobs.

Characteristics:

- The local resource management systems have to have advance reservation capabilities.
- The Grid scheduler must support the co-allocation of resources.
- An information service is needed. Such a service is not mandatory, but is it impractical for the Grid scheduler to query all potential resources individually.
- A co-allocation scenario may require a negotiation framework (like e.g. one based on WS-Agreement [33]) since it is likely that a simple two-step protocol to reserve resources does not provide a solution (especially if the number of resources involved grows).

3.5 Complex workflow

Scenario:
A job in this scenario comprises a complex workflow which generally includes different steps to be executed on different resources. Scheduling such a complex workflow requires advance reservation of resources with different allocation times.

This implies that the Grid scheduler has to take the dependencies between the different resource requests into account. The "Scheduling complex workflows" use case (see Section 2.1) provides a generic description of such a scenario.

Characteristics:

- The local resource management systems have to have advance reservation capabilities.
- The Grid scheduler must support the co-allocation of resources.
- Information service and negotiation framework are needed.
- A workflow can be static or dynamic with respect to some of its properties. Dynamic workflows may be adapted to the current state of job execution.

3.6 Data Staging

Scenario:
This is a specific usage pattern where the job/workflow requires some data to be made available before the job or parts of the workflow are executed. To enable this, the scheduler needs to check the availability of the data and it must consider the time needed to transfer the data to where the job/workflow expects it to be.

Characteristics:

- The Grid scheduler needs to synchronise the job/workflow execution with the staging of data.

4 References

[1] Grid Scheduling Architecture Research Group (GSA-RG), web site, 2005. Online: https://forge.gridforum.org/projects/gsa-rg/.
[2] Grid Resource Allocation Agreement Protocol Working Group (GRAAP-WG), web site, 2005. Online: https://forge.gridforum.org/projects/graap-wg/.
[3] OASIS Web Services Resource Framework (WSRF) TC, web site, 2005. Online: http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsrf.
[4] OGSA Resource Usage Service Working Group (RUS-WG), web site, 2005. Online: https://forge.gridforum.org/projects/rus-wg/.
[5] OASIS Web Services Notification TC, web site, 2005. Online: http://www.oasis-open.org/committees/wsn/charter.php.
[6] M. Gudgin, M. Hadley, N. Mendelsohn, J. Moreau, H.F. Nielsen. SOAP Version 1.2 Part 1: Messaging Framework. W3C Recommendation, W3C, 24 June, 2003. Online: http://www.w3.org/TR/soap12-part1/.
[7] Gokhale, Kumar, and Sahuguet. Reinventing the Wheel? CORBA vs. Web Services. Online: http://www2002.org/CDROM/alternate/395/.
[8] OASIS Web Services Distributed Management TC, web site, 2005. Online: http://www.oasis-open.org/committees/wsdm/charter.php.
[9] M. Aldinucci, S. Campa, M. Coppola, M. Danelutto, D. Laforenza, D. Puppin, L. Scarponi, M. Vanneschi, and C. Zoccolo. Components for High-Performance Grid Programming in Grid.it. In Component Models and Systems for Grid Applications, V. Getov and T. Kielmann, eds., Springer, 2004.
[10] M. Vanneschi. The Programming Model of ASSIST, an Environment for Parallel and Distributed Portable Applications. Parallel Computing, 28(12), 2002.
[11] M. Aldinucci, S. Campa, P. Ciullo, M. Coppola, S. Magini, P. Pesciullesi, L. Potiti, R. Ravazzolo, M. Torquati, M. Vanneschi, and C. Zoccolo. The Implementation of ASSIST, an Environment for Parallel and Distributed Programming. In Proc. of the Euro-Par 2003 Conference: Parallel Processing, H. Kosch, L. Böszörményi and H. Hellwagner, eds., LNCS 2970, pp. 712-721, Springer, 2003.
[12] GridCCM, web site, 2005. Online: http://www.irisa.fr/paris/Gridccm/.
[13] ProActive, web site, 2005. Online: http://www-sop.inria.fr/oasis/ProActive/.
[14] The Knowledge Grid Lab, web site, 2005. Online: http://dns2.icar.cnr.it/kgrid/.
[15] M. Cannataro and D. Talia. The Knowledge Grid. Communications of the ACM, 46(1):89-93, 2003.
[16] A. Pugliese and D. Talia. Application-oriented scheduling in the Knowledge Grid: a model and architecture. International Conference on Computational Science and its Applications (ICCSA), LNCS 3044, pp. 55-65, Springer, Berlin, Germany, 2004.
[17] M. Cannataro, A. Congiusta, A. Pugliese, D. Talia, and P. Trunfio. Distributed data mining on Grids: services, tools, and applications. IEEE Transactions on Systems, Man, and Cybernetics: Part B (TSMC-B). To appear.
[18] V. Welch, F. Siebenlist, I. Foster, J. Bresnahan, K. Czajkowski, J. Gawor, C. Kesselman, S. Meder, L. Pearlman, S. Tuecke. Security for Grid Services. Twelfth International Symposium on High Performance Distributed Computing (HPDC-12), IEEE Press, 2003.
[19] Globus Toolkit 3.0 Documentation, web site, 2006. Online: http://www.globus.org/toolkit/docs/3.0/

[20] The GridWay Project, web site, 2006. Online: www.gridway.org.
[21] The DRMAA Working Group, web site, 2006. Online: www.drmaa.org.
[22] The Globus Alliance, web site, 2006. Online: www.globus.org.
[23] J. Schopf. Ten Actions when Superscheduling. Grid Forum Document GFD.4, Global Grid Forum, 2001. http://www.ggf.org/documents/GFD.4.pdf.
[24] E. Huedo, R. S. Montero, and I. M. Llorente. A Framework for Adaptive Execution on Grids. Software: Practice and Experience 34(7), pp. 631-651, 2004.
[25] Gram RSL Schema Documentation, web site, 2006. Online: http://www.globus.org/toolkit/docs/3.0/gram/rsl-schema.html.
[26] M. Dalheimer, F.-J. Pfreundt, and P. Merz. Calana – A General-purpose Agent-based Grid Scheduler. In Proceedings of the 14th IEEE International Symposium on High Performance Distributed Computing (HPDC-14). To appear.
[27] A. Hoheisel and U. Der. An XML-based Framework for Loosely Coupled Applications on Grid Environments. In P.M.A. Sloot, editor, ICCS 2003, LNCS 2657, pp. 245-254. Springer, 2003.
[28] A. Hoheisel. User tools and languages for graph-based grid workflows. In Grid Workflow 2004, Special Issue of Concurrency and Computation: Practice and Experience, 2004.
[29] IETF RFC 3920 - Extensible Messaging and Presence Protocol (XMPP).
[30] VIOLA – Vertically Integrated Optical Testbed for Large Application in DFN. Project web site, 2005. Online: http://www.viola-testbed.de/.
[31] D. Erwin, ed. UNICORE Plus Final Report – Uniform Interface to Computing Resources. UNICORE Forum e.V., ISBN 3-00-011592-7, 2003.
[32] O. Wäldrich, Ph. Wieder, and W. Ziegler. A Meta-Scheduling Service for Co-allocating Arbitrary Types of Resources. In Proceedings of the Sixth International Conference on Parallel Processing and Applied Mathematics, 2005, Grid Resource Management Workshop. To appear.
[33] 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. Grid Forum Draft, Version 1.0, Global Grid Forum, 2005.

5 Editor Information

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

Ramin Yahyapour
Computer Engineering Institute
University Dortmund
44221 Dortmund, Germany
Ramin.yahyapour@udo.edu

Contributing authors:
Andrea Pugliese, Domenico Talia
DEIS-University of Calabria
Via P. Bucci, 41/C, Rende ,Italy
{apugliese,talia}@deis.unical.it

Jaegyoon Hahm
Supercomputing Center
Korea Institute of Science and Technology Information
305-333, Daejeon, Korea
Phone: +82-42-869-0580,
jaehahm@kisti.re.kr

Ignacio M. Llorente
Dpto. Arquitectura de Computadores y Automatica
Facultad de Informatica
Universidad Complutense
llorente@dacya.ucm.es
http://asds.dacya.ucm.es/

Nicola Tonellotto
Istituto di Scienza e Tecnologie dell´Informazione “A. Faedo“
Consiglio Nazionale delle Ricerche
Via G. Moruzzi 1, 56100, Pisa, Italy
nicola.tonellotto@isti.cnr.it

Mathias Dalheimer
Fraunhofer Institut fuer Techno- und Wirtschaftsmathematik (ITWM)
67657 Kaiserslautern, Germany
dalheimer@itwm.fraunhofer.de

Wolfgang Ziegler
Fraunhofer-Institute for Algorithms and Scientific Computing (SCAI)
Schloss Birlinghoven,
D-53754 Sankt Augustin, Germany
Wolfgang.Ziegler@scai.fraunhofer.de

6 Full Copyright Notice

7 Intellectual Property Statement