

A Meta-Scheduling Service for Co-allocating Arbitrary Types of Resources

O. Wäldrich

`oliver.waeldrich@scai.fraunhofer.de`

Fraunhofer Institute SCAI

Department for Web-based Applications

53754 Sankt Augustin, Germany

Ph. Wieder

`ph.wieder@fz-juelich.de`

Central Institute for Applied Mathematics

Research Centre Jülich

52425 Jülich, Germany

W. Ziegler

`wolfgang.ziegler@scai.fraunhofer.de`

Fraunhofer Institute SCAI

Department for Web-based Applications

53754 Sankt Augustin, Germany



**CoreGRID Technical Report
Number TR-0010**

December 2, 2005

Institute on Resource Management and Scheduling

CoreGRID - Network of Excellence

URL: <http://www.coregrid.net>

A Meta-Scheduling Service for Co-allocating Arbitrary Types of Resources

O. Wäldrich

`oliver.waeldrich@scai.fraunhofer.de`

Fraunhofer Institute SCAI

Department for Web-based Applications
53754 Sankt Augustin, Germany

Ph. Wieder

`ph.wieder@fz-juelich.de`

Central Institute for Applied Mathematics
Research Centre Jülich
52425 Jülich, Germany

W. Ziegler

`wolfgang.ziegler@scai.fraunhofer.de`

Fraunhofer Institute SCAI

Department for Web-based Applications
53754 Sankt Augustin, Germany

CoreGRID TR-0010

December 2, 2005

Abstract

The Grid paradigm implies the sharing of a variety of resources across multiple administrative domains. In order to execute a work-flow using these distributed resources an instrument is needed to co-allocate resources by reaching agreements with the different local scheduling systems involved. Apart from compute resources to execute the work-flow the co-ordinated usage of other resource types must be also guaranteed, as there are for example a network connectivity with dedicated QoS parameters or a visualisation device. We present a Web Service-based MetaScheduling Service which allows to negotiate a common time slot with local resource management systems to enable the execution of a distributed work-flow. The successful negotiation process results in a formal agreement based on the WS-Agreement recommendation that is currently specified by the GRAAP working group of the Global Grid Forum. As a use case we demonstrate the integration of this MetaScheduling Service into the UNICORE middleware.

1 Introduction

To successfully execute distributed applications or work-flows, usually different resources like compute nodes, visualisation devices, storage devices, or network connectivity with a defined QoS are required at the same time or in a certain sequence. Orchestrating such resources locally within one organisation represents only a minor task, whereas the orchestration of resources on a Grid level requires a service that is able to solve the same problems in an environment that may stretch across several administrative domains. Additional conditions have to be taken into account, like the compliance with site-specific policies or the protection of a site's autonomy.

In this paper we first describe in Section 2 an environment where co-allocation of resources is of vital importance, the requirements for the MetaScheduling Service that provides the required co-allocation means, and related work. In

a next step we characterise the functionality of the MetaScheduling Service (Section 3), followed by a description of the current implementation. Then, in Section 5, we present the integration of the scheduling system into the UNICORE Grid middleware [1]. The performance of the the whole system is evaluated in Section 6, and the last section contains conclusions and an outlook to future work.

2 Scope, Requirements, and Related Work

The subject we are dealing with is scheduling and resource management in Grids and our primary focus is the co-allocation of different resources needed for the execution of a distributed application or a work-flow. Today's Grid-aware applications benefit more and more from using heterogeneous hardware that allows to optimise the performance of the applications. However, to make use of such distributed resources a tool or service is required that is capable of dealing with multiple policies for the operation and usage of resources. Furthermore, the scheduling systems that manage these resources will usually not be altered when the resources, in addition to local utilisation, are made available for usage in a Grid environment. Taking into account heterogeneity, site autonomy, and different site policies, a common approach for co-scheduling [2] of resources is not available so far. However, there have been several approaches over the last years ranging from commercial products like those from the Load Sharing Facility (LSF) family [3], to project-specific implementations like MARS [4] (a meta-scheduler for Campus Grids), GridFlow [5] (supporting work-flows in the DataGrid), or developments of the GrADS or the GriPhyN project. Other approaches like the one in Condor-G [6] or Legion's JobQueue [7] are limited to the underlying middleware environment. The Community Scheduler Framework (CSF) [8] is a global scheduler that interfaces with local schedulers such as OpenPBS [9], LSF or SGE [10], but it requires the obsolete Globus Toolkit 3.0 [11]. Finally there is GARA [12], an API supporting co-scheduling of, inter alia, network resources in a Globus middleware environment.

All the approaches mentioned above are to some extent limited and not suitable to be adopted to the needs of a common meta-scheduling service, because of

- their commercial nature,
- their limited support for different types of resources,
- their restriction to the needs of a specific project,
- their limitation to a particular middleware environment, and
- the use of proprietary protocols.

The work presented in the following sections of the paper tries to overcome the limitations by providing an extensible service that is not restricted to the needs of a particular project, that implements evolving standards, and that is able to support arbitrary types of resources.

3 Required Functionality of the MetaScheduling Service

To achieve co-allocation of resources managed by multiple, usually different scheduling systems, the minimal requirement these systems have to fulfil is to provide functions to

1. schedule a single reservation some time in the future (e.g. "from 5:00 pm to 8:00 pm tomorrow") and to
2. give an aggregated overview of the usage of the managed resources between now and a defined time in future.

Once a reservation is scheduled the starting time for it is fixed, i.e. it may not change except for the reservation being cancelled. This feature is called advance reservation. There are at least two possibilities to realise such advance reservation. The first possibility is to schedule a reservation for a requested time, called fixed time scheduling. The second possibility is to schedule a reservation not before a given time, which means a scheduling system tries to place the reservation at the requested time, otherwise it will be scheduled for the earliest possible time after the one requested. This results in a first fit reservation. The implementation of the MetaScheduling Service described in this paper interacts with, but is not limited to, scheduling systems that implement the first fit behaviour. The main function

of the MetaScheduling Service is to negotiate the reservation of network-accessible resources that are managed by their respective local scheduling systems. The goal of the negotiation is to determine a common time slot where all required resources are available for the requested starting time of the job. The major challenges for a meta-scheduler are

- to find Grid resources suitable for the user's request,
- to take security issues like user authentication and authorisation into account,
- to respect the autonomy of the sites offering the resources, and
- to cope with the heterogeneity of the local scheduling systems.

We do not address the problem of finding suitable resources here, this task is usually delegated to a Grid information system or a resource broker. Security issues are only considered here with respect to user authentication and authorisation as the MetaScheduling Service has to reserve on behalf of the user's respective identity at the sites that he wants to use resources from. The implementation described in the next section addresses both site autonomy and heterogeneity.

4 Implementation of the MetaScheduling Service

To interact with different types of scheduling systems we decided to use the adapter pattern approach. The role of such adapters is to provide a single interface to the MetaScheduling Service by encapsulating the specific interfaces of the different local scheduling systems. Thus the MetaScheduling Service can negotiate resource usage using a single interface. The adapters are connected to both the MetaScheduling service and the scheduling systems and may therefore be installed either at the site hosting the MetaScheduling Service, the (remote) sites where the scheduling systems are operated, or at any other system accessible through a network. Currently adapters are available for two scheduling systems: EASY and a proprietary scheduling system. Two more adapters will be available late summer 2005: one for the Portable Batch System Professional (PBS Pro) and another one for ARGON, the resource management system for network resources that is currently under development in the VIOLA project [16].

4.1 Negotiation protocol

In this section we describe the protocol the MetaScheduling Service uses to negotiate the allocation of resources with the local scheduling systems. This protocol is illustrated in Fig. 1. The user specifies the duration of the meta-job and additionally - for each subsystem - reservation characteristics like the number of nodes of a cluster or the bandwidth of the connections between nodes. In the UNICORE based VIOLA testbed the UNICORE client is used to describe the request of the user. The client sends the job description to the MetaScheduling Service using the Web Services Agreement (WS-Agreement) protocol [14]. Based on the information in the agreement the MetaScheduling Service starts the resource negotiation process:

1. The MetaScheduling Service queries the adapters of the selected local systems to get the earliest time the requested resources will be available. This time possibly has to be after an offset specified by the user.
2. The adapters acquire previews of the resource availability from the individual scheduling systems. Such a preview comprises a list of time frames during which the requested QoS (e.g. a fixed number of nodes) can be provided. It is possible that the preview contains only one entry or even zero entries if the resource is fully booked within the preview's time frame. Based on the preview the adapter calculates the possible start-time.
3. The possible start times are sent back to the MetaScheduling Service.
4. If the individual start times allow the co-allocation of the resources, the respective resources are reserved via the Adapters of the local schedulers. If the individual start times do not allow the co-allocation of the resources, the MetaScheduling Service uses the latest possible start time as the earliest start time for the next scheduling iteration. The process is repeated from step 1. until a common time frame is found or the end of the preview period for all the local systems is reached. The latter case generates an error condition.

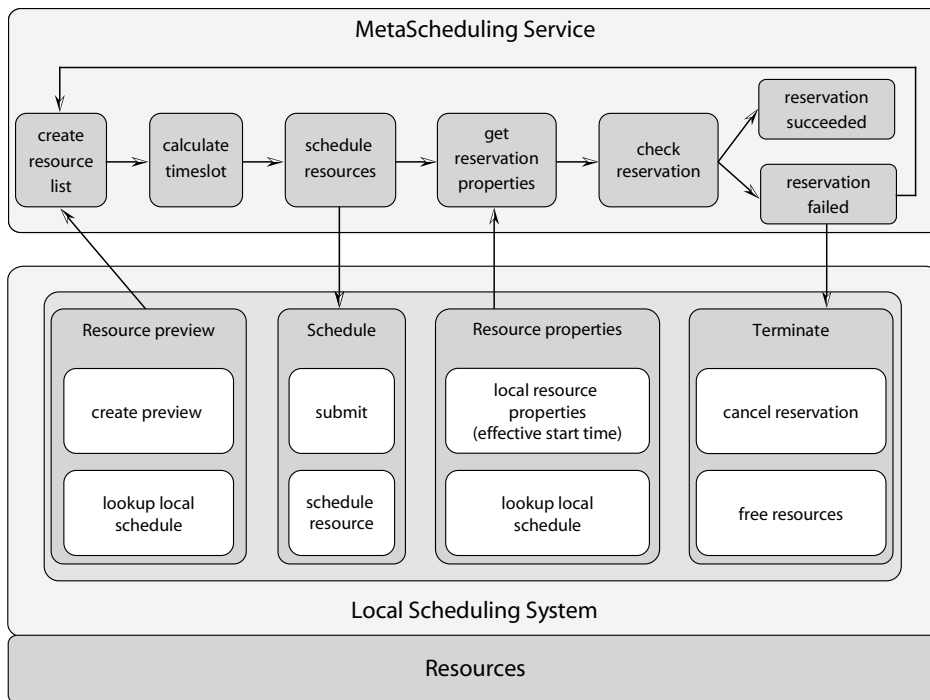


Figure 1: The negotiation process

5. In case the individual start times match, the MetaScheduling Service checks the scheduled start times for each reservation by asking the local schedulers for the properties of the reservation. This step is necessary because in the meantime new reservations may have been submitted by other users or processes to the local schedulers, preventing the scheduling of the reservation at the requested time.
6. If the MetaScheduling Service detects one or more reservations that are not scheduled at the requested time, all reservations will be cancelled. The latest effective start time of all reservations will be used as the earliest start time to repeat the process beginning with step 1.
7. If all reservations are scheduled for the appropriate time the co-allocation of the resources has been completed.
8. The IDs of the MetaScheduling Service and the local reservations are added to the agreement and a reference to the agreement is sent back to the client which initially submitted the meta-job (in case of the VIOLA testbed it is the UNICORE client).

4.2 Hosting Environment and Command Line Interface

For the hosting environment of the MetaScheduling Service we use Sun Java 1.3.1 JRE with Apache Tomcat 3.3.2 as a servlet engine. In addition we make use of the Apache SOAP Framework 2.3 for sending and receiving SOAP [13] messages. The interface between the adapter and the local scheduling systems is implemented as a simple CGI module, which is hosted at the Apache web server and wraps the local scheduling system's commands. The communication between Apache and the adapters is secured by using HTTPS.

A simple Command Line Interface (CLI) is also available to access the MetaScheduling Service. It enables users to submit a meta-scheduling request, to query job details related to a certain reservation, and to cancel a given reservation. To submit a request a user may specify the duration of the reservation, the resources needed per reservation (number of nodes, network connectivity, ...), and the executable for each site.

The current CLI implementation does not contain integrated security means to provide single sign-on, as e.g. the UNICORE client does. Instead the CLI user has to enter a valid login and password for every requested resource. The credentials are stored by the CLI, passed to the MetaScheduling Service, and used for authentication at the local

systems. Please note that this is an interim solution and that the meta-scheduling framework will provide a concise security solution at a later implementation stage.

4.3 Interface between MetaScheduling Service and Adapter

The interface/protocol between the MetaScheduling Service and the adapters is implemented using Web Services and it basically provides the five functions `couldRunAt()`, `submit()`, `submit()`, `cancel()`, `state()`, and `bind()`. Please refer to Appendix A for a detailed specification of the functions and examples of the SOAP messages being sent.

These functions cover the whole negotiation process described in Section 4.1. The Web Services interface/protocol as implemented is not standardised, but the GRAAP [15] working group at the Global Grid Forum (GGF) is currently discussing to address the standardisation of a protocol for the negotiation process. Once such a recommendation is available we plan to implement it instead of the current one.

4.4 Implementation of the Agreement

The development of the MetaScheduling Service includes, as already mentioned in Section 4.1, an implementation of the WS-Agreement specification [14]. The client submits a request using an agreement offer (a template provided by the MetaScheduling Service and filled in with the user's requirements by the client) to the MetaScheduling Service. It then negotiates the reservation of the resources in compliance with the QoS defined in the agreement template. As a result of the successfully completed negotiation process a valid agreement is returned to the client, containing the scheduled end times of the reservation, the reservation identifier, and further information.

5 Integration into the UNICORE Middleware

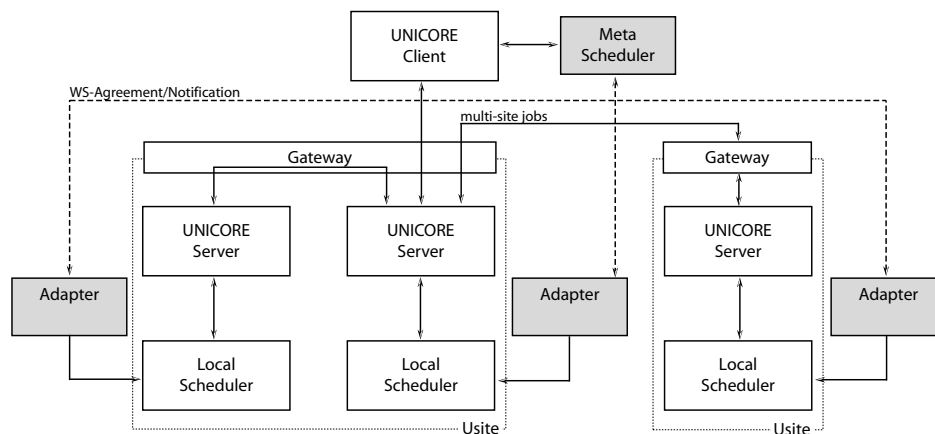


Figure 2: The meta-scheduling architecture

The first generation meta-scheduler architecture developed in the VIOLA project [16] focuses on the scheduling functionality requiring only minimal changes to the UNICORE system (please refer to [1] for an in-depth description of the UNICORE models and components). As depicted in Fig. 2, the system comprises the adapter, the MetaScheduling Service itself [17], and a MetaScheduling plug-in (which is part of the client and not pictured separately). Before submitting a job to a UNICORE Site (Usite), the MetaScheduling plug-in and the MetaScheduling Service exchange the data necessary to schedule the resources needed. The MetaScheduling Service is then (acting as an Agreement Consumer in WS-Agreement terms [14]) contacting the adapter (acting as an Agreement Manager) to request a certain level of service, a request which is translated by the Manager into the appropriate local scheduling system commands. In case of VIOLA's computing resources the targeted system is the EASY scheduler. Once all resources are reserved at the requested time the MetaScheduling Service notifies the UNICORE Client via the MetaScheduling plug-in to submit

the job. This framework will also be used to schedule the interconnecting network, but potentially any resource can be scheduled if a respective adapter/Agreement Manager is implemented and the MetaScheduling plug-in generates the necessary scheduling information. The follow-on generation of the meta-scheduling framework will then be tightly integrated into UNICORE.

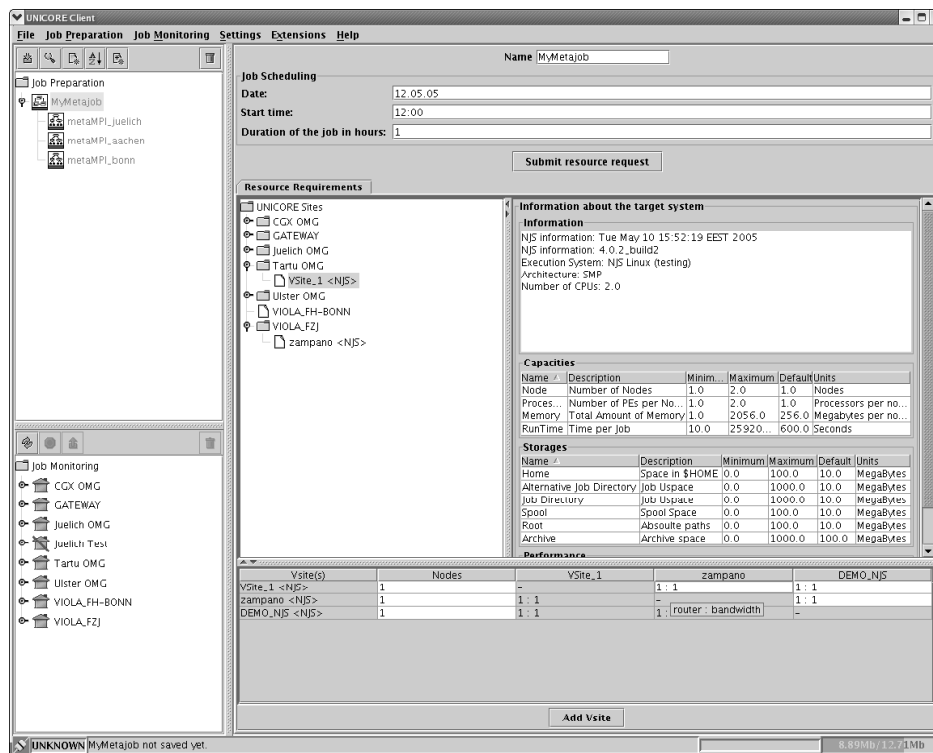


Figure 3: The UNICORE client with meta-scheduling extensions

6 Experiences and Performance Evaluation

We set up different tests to evaluate the performance of the MetaScheduling Service. All tests were performed on a PC Cluster where multiple local scheduling systems and MetaScheduling services were deployed across the compute nodes. We executed the following two types of performance tests:

1. Reservations with multiple adapter instances (Fig. 4).
2. Reservations with multiple MetaScheduling Service instances (Fig. 5).

In the first test series we used a single MetaScheduling Service instance to co-allocate resources across six different local schedulers. This implied that six partial reservations were generated for six different local scheduling instances that were randomly chosen from a pool of 160 instances. The queue of each local scheduler was randomly initialised to simulate a certain load per system. In the second test series we realised a layered approach: The MetaScheduling Service that received the reservation request from the user (the topMSS) delegated the reservations to other MetaScheduling Services which then negotiated with the local schedulers. As in the first test a reservation consisted of six partial reservations. Apart from the topMSS three MetaScheduling Services were used, each being responsible for the co-allocation of resources managed by two local schedulers. The topMSS instance negotiated the complete reservation with these three instances which for the topMSS take the role of adapters as in the first test using the same protocol for the inter-MetaScheduling Service communication.

Both tests indicate that the co-allocation approach selected is reasonable: a complete schedule is delivered in less than 20 seconds (30 seconds in the case of layered MetaScheduling Services). Fig. 4 and 5 indicate a slight increase

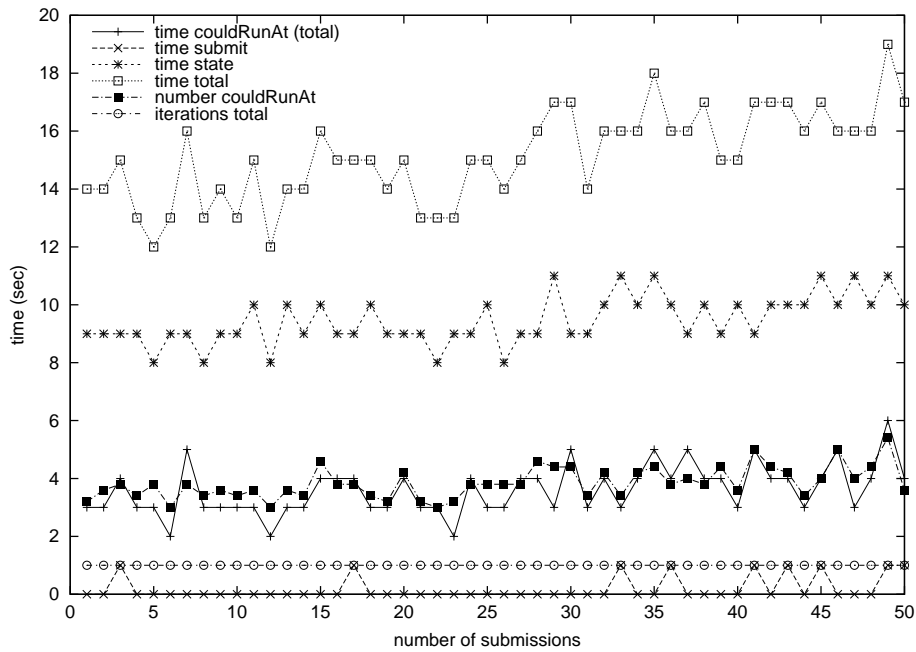


Figure 4: Performance of a MetaScheduling Service negotiating with multiple adapters

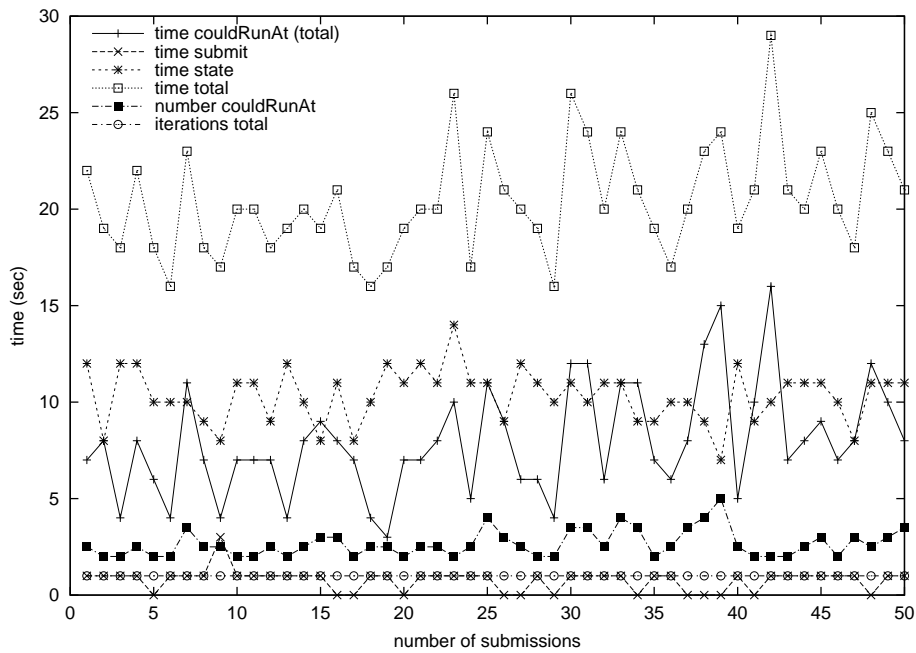


Figure 5: Performance of a MetaScheduling Service negotiating with multiple MetaScheduling Services

of the total reservation time in case the local systems' load increases and thus the number of lookups needed for identifying free slots and for scheduled reservations also increases. The total number of iterations indicates how often the co-allocation process had to be restarted due to reservation inconsistencies at the local schedulers (see Section 4.1, step 7.). In the two tests series this value was constant since there were no additional scheduling activities (apart from

those initiated by the MetaScheduling Service) carried out that could interfere.

7 Conclusions and future work

In this paper a meta-scheduling framework has been described which is capable of co-allocating arbitrary types of resources by means of adapters that abstract the different interfaces of the local scheduling systems. An implementation has been presented that may be used via a command line interface or integrated into a Grid middleware. As a use case a first approach to an WS-Agreement-based integration into the UNICORE Grid system has been shown. Future work will concentrate on several improvements of the service and focus, e.g. on:

- implementing adapters for additional scheduling systems,
- implementing the WS-Negotiation protocol as it is evolving at the GGF,
- integrating the MetaScheduling Service into OGSA/WSRF-based UNICORE [18] and Globus [19] systems,
- providing resource broker capabilities,
- integrating the Intelligent Scheduling System (ISS) for selection of resources best fitted to execute an specific application [21], and
- integrating a Grid Scheduling Ontology [20].

8 Acknowledgements

This paper includes work carried out jointly within the CoreGRID Network of Excellence funded by the European Commission's IST programme under grant #004265. Some of the work reported in this paper is funded by the German Federal Ministry of Education and Research through the VIOLA project under grant #123456.

References

- [1] D. Erwin, ed. *UNICORE Plus Final Report – Uniform Interface to Computing Resources*. UNICORE Forum e.V., ISBN 3-00-011592-7, 2003.
- [2] J. Schopf. Ten Actions when Grid Scheduling. In *Grid Resource Management* (J. Nabrzyski, J. Schopf and J. Weglarz, eds.), pages 15-23, Kluwer Academic Publishers, 2004.
- [3] Load Sharing Facility, Resource Management and Job Scheduling System. Web site, 22 November 2005, <<http://www.platform.com/Products/Platform.LSF.Family/>>.
- [4] A. Bose, B. Wickman, and C. Wood. MARS: A Metascheduler for Distributed Resources in Campus Grids. In *5th International Workshop on Grid Computing (GRID 2004)*. IEEE Computer Society, 2004.
- [5] J. Weinberg, A. Jagatheesan, A. Ding, M. Faerman, and Y. Hu. Gridflow: Description, Query, and Execution at SCEC using the SDSC Matrix In *13th IEEE International Symposium on High Performance Distributed Computing (HPDC'04)* IEEE Computer Society, 2004.
- [6] D. Thain and M. Livny. Building Reliable Clients and Servers. In *The Grid: Blueprint for a New Computing Infrastructure* (I. Foster and C. Kesselman, eds.), Morgan Kaufmann, 2003.
- [7] D. Katramatos, M. Humphrey, A. S. Grimshaw, and S. J. Chapin. JobQueue: AComputational Grid-Wide Queuing System. In *Grid Computing - GRID 2001, Second International Workshop*, Volume 2242 of Lecture Notes in Computer Science, Springer, 2001.
- [8] Community Scheduler Framework. Web site, 22 November 2005, <<http://sourceforge.net/projects/gscf>>.

- [9] OpenPBS Batch Processing and Resource Management System. Web site, 22 November 2005, <<http://www.openpbs.org/>>.
- [10] Sun Grid Engine. Web site, 22 November 2005, <<http://www.sun.com/software/gridware/>>.
- [11] The Globus Toolkit 3.0. Web site, 22 November 2005, <<http://www.globus.org/toolkit/downloads/3.0/>>.
- [12] I. Foster, A. Roy, and V. Sander. A Quality of Service Architecture that Combines Resource Reservation and Application Adaptation. In *8th International Workshop on Quality of Service (IWQOS 2000)*, pages 181-188, June, 2000.
- [13] Simple Object Access Protocol Specification, SOAP Specification version 1.2. Web site, 2005. Online: <http://www.w3.org/TR/soap12/>.
- [14] 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 2005/09, Global Grid Forum, September, 2005.
- [15] Grid Resource Allocation Agreement Protocol Working Group. Web site, 22 November 2005, <<https://forge.gridforum.org/projects/graap-wg/>>.
- [16] VIOLA – Vertically Integrated Optical Testbed for Large Application in DFN. Web site, 22 November 2005, <<http://www.viola-testbed.de/>>.
- [17] G. Quecke and W. Ziegler. MeSch – An Approach to Resource Management in a Distributed Environment. In *Proc. of 1st IEEE/ACM International Workshop on Grid Computing (Grid 2000)*, Volume 1971 of Lecture Notes in Computer Science, pages 47-54, Springer, 2000.
- [18] R. Menday and Ph. Wieder. GRIP: The Evolution of UNICORE towards a Service-Oriented Grid. In *Proc. of the 3rd Cracow Grid Workshop (CGW'03)*, Cracow, PL, Oct. 27-29, 2003.
- [19] The Globus Toolkit 4.0. Project documentation web site, 22 November 2005, <<http://www.globus.org/toolkit/docs/4.0/>>.
- [20] Ph. Wieder and W. Ziegler. Bringing Knowledge to Middleware - The Grid Scheduling Ontology. In *First Core-GRID Workshop*, Springer, 2005, to appear.
- [21] V. Keller, K. Cristiano, R. Gruber, P. Kuonen, S. Maffioletti, N. Nellari, M.-C. Sawley, T.-M. Tran, Ph. Wieder, and W. Ziegler. Integration of ISS into the VIOLA Meta-scheduling Environment. In *Proc. of the Integrated Research in Grid Computing Workshop*, Pisa, IT, November 28-20, 2005, to appear.

A Interface between MetaScheduling Service and Adapter

The interface between the MetaScheduling Service and the Adapter is defined by the five functions `couldRunAt()`, `submit()`, `cancel()`, `state()`, and `bind()` (see also Section 4.3). These functions are used to execute the steps in the negotiation protocol which require communication with the local schedulers, a description of which can be found in Section 4.1. The `bind()` function is exclusively used to communicate with local network schedulers. In the following subsections for each of these functions

- the function call,
- the purpose of the function,
- the attributes,
- the return value, and
- a sample SOAP request and response message

are listed.

A.1 `couldRunAt()`

Call: `Date couldRunAt(Job job_description)`
Purpose: Get the earliest possible runtime for a job.
Attributes: Description of the job.
Return value: The earliest possible runtime for the job on the system the respective request has been sent to.

Listing 1: `couldRunAt()` – sample SOAP request

```
<?xml version='1.0' encoding='UTF-8?'><SOAP-ENV:Envelope
xmlns:SOAP-ENV='http://schemas.xmlsoap.org/soap/envelope/'
xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
xmlns:xsd='http://www.w3.org/2001/XMLSchema'><SOAP-ENV:Body>
<ns1:couldRunAt xmlns:ns1='urn:SchedulerServer'><SchedulerServer
xsi:type='ns1:SchedulerServer'
SOAP-ENV:encodingStyle='http://schemas.xmlsoap.org/soap/encoding'>
<IID xsi:type='xsd:string' xsi:nil='true'></interactive
xsi:type='xsd:boolean'>false</interactive><jobType
xsi:type='xsd:int'>0</jobType><nodes
xmlns:ns2='http://schemas.xmlsoap.org/soap/encoding/'
xsi:type='ns2:Array' ns2:arrayType='xsd:string[0]'></nodes>
<nodesRequired xsi:type='xsd:int'>1</nodesRequired>
<requestedStartTime
xsi:type='xsd:dateTime'>2004-07-27T17:12:00.000Z</requestedStartTime>
<scheduledStartTime
xsi:type='xsd:dateTime'>2004-07-27T17:07:16.953Z</scheduledStartTime>
<shellScript xsi:type='xsd:string'>test.sh</shellScript><state
xsi:type='xsd:int'>-1</state><timeRequired
xsi:type='xsd:int'>20</timeRequired><usingAdvancedReservation
xsi:type='xsd:boolean'>true</usingAdvancedReservation>
</SchedulerServer></ns1:couldRunAt></SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Listing 2: `couldRunAt()` – sample SOAP response

```
<?xml version='1.0' encoding='UTF-8?'>
<SOAP-ENV:Envelope xmlns:SOAP-ENV='http://schemas.xmlsoap.org/soap/envelope/'
xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
xmlns:xsd='http://www.w3.org/2001/XMLSchema'>
<SOAP-ENV:Body>
<ns1:couldRunAtResponse xmlns:ns1='urn:SchedulerServer' SOAP-ENV:encodingStyle='http://schemas.xmlsoap.org/soap/encoding'>
<return xsi:type='xsd:dateTime'>2004-07-27T19:17:00.000Z</return>
</ns1:couldRunAtResponse>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

A.2 submit()

Call: String submit(Job job_description)
Purpose: Submit a job to a local scheduler. The submit() call results in an advance reservation of the requested resource.
Attributes: Description of the job.
Return value: A job ID unique within the meta-scheduling environment.

Listing 3: submit() – sample SOAP request

```
<?xml version='1.0' encoding='UTF-8?'>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <SOAP-ENV:Body>
    <ns1:submit xmlns:ns1="urn:SchedulerServer">
      <SchedulerServer xsi:type="ns1:SchedulerServer" SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
        <ID xsi:type="xsd:string" xsi:nil="true"/>
        <interactive xsi:type="xsd:boolean">false</interactive>
        <jobType xsi:type="xsd:int">0</jobType>
        <nodes xmlns:ns2="http://schemas.xmlsoap.org/soap/encoding/" xsi:type="ns2:Array" ns2:arrayType="xsd:string[0]"></nodes>
        <nodesRequired xsi:type="xsd:int">3</nodesRequired>
        <requestedStartTime xsi:type="xsd:dateTime">2004-07-27T22:05:00.000Z</requestedStartTime>
        <scheduledStartTime xsi:type="xsd:dateTime">2004-07-27T17:07:16.953Z</scheduledStartTime>
        <shellScript xsi:type="xsd:string">test.sh</shellScript>
        <state xsi:type="xsd:int">-1</state>
        <timeRequired xsi:type="xsd:int">20</timeRequired>
        <usingAdvancedReservation xsi:type="xsd:boolean">true</usingAdvancedReservation>
      </SchedulerServer>
    </ns1:submit>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Listing 4: submit() – sample SOAP response

```
<?xml version='1.0' encoding='UTF-8?'>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <SOAP-ENV:Body>
    <ns1:submitResponse xmlns:ns1="urn:SchedulerServer" SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
      <return xsi:type="xsd:string">072719154808</return>
    </ns1:submitResponse>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

A.3 cancel()

Call: boolean cancel(String job_id)
Purpose: Cancel a job with a given job ID.
Attributes: The ID of the job which has to be cancelled.
Return value: Success or failure.

Listing 5: cancel() – sample SOAP request

```
<?xml version='1.0' encoding='UTF-8?'>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <SOAP-ENV:Body>
    <ns1:cancel xmlns:ns1="urn:SchedulerServer">
      <SchedulerServer xsi:type="xsd:string" SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">072719153196</SchedulerServer>
    </ns1:cancel>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Listing 6: cancel() – sample SOAP response

```
<?xml version='1.0' encoding='UTF-8?'>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <SOAP-ENV:Body>
    <ns1:cancelResponse xmlns:ns1="urn:SchedulerServer" SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
      <result xsi:type="xsd:boolean">true</result>
    </ns1:cancelResponse>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

A.4 state()

Call: Job state(String job_id)
Purpose: Retrieve the status of a job.
Attributes: The ID of the job.
Return value: The complete job is returned including the scheduled start time.

Listing 7: state() – sample SOAP request

```
<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <SOAP-ENV:Body>
    <ns1:state xmlns:ns1="urn:SchedulerServer">
      <SchedulerServer xsi:type="xsd:string" SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">072719154808
    </SchedulerServer>
    </ns1:state>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Listing 8: state() – sample SOAP response

```
<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <SOAP-ENV:Body>
    <ns1:stateResponse xmlns:ns1="urn:SchedulerServer" SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
      <return xsi:type="ns1:SchedulerServer">
        <timeRequired xsi:type="xsd:int">20</timeRequired>
        <state xsi:type="xsd:int">1</state>
        <nodes xmlns:ns2="http://schemas.xmlsoap.org/soap/encoding/" xsi:type="ns2:Array" ns2:arrayType="xsd:string[0]"></nodes>
        <jobType xsi:type="xsd:int">0</jobType>
        <requestedStartTime xsi:type="xsd:dateTime">2004-07-28T03:08:00.000Z</requestedStartTime>
        <shellScript xsi:type="xsd:string">test.sh</shellScript>
        <nodesRequired xsi:type="xsd:int">1</nodesRequired>
        <scheduledStartTime xsi:type="xsd:dateTime">2004-07-28T03:08:00.000Z</scheduledStartTime>
        <usingAdvancedReservation xsi:type="xsd:boolean">>false</usingAdvancedReservation>
        <JID xsi:type="xsd:string" xsi:nil="true"/>
        <interactive xsi:type="xsd:boolean">>false</interactive>
      </return>
    </ns1:stateResponse>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

A.5 bind()

Since the local network scheduler also developed in the VIOLA project is not yet integrated into the meta-scheduling environment, the `bind()` function is not yet fully specified.

Call: `boolean bind(AddressList address_list)`
Purpose: Publicize source and destination addresses of the network connections belonging to a certain reservation at run time. This is necessary since the addresses of the nodes are potentially not known until runtime but they are needed to guarantee a job exclusive access to the network bandwidth.
Attributes: List of source and destination IP addresses for each site-to-site connection.
Return value: Success or failure.

Listing 9: `bind()` – sample SOAP request

```
<?xml version='1.0' encoding='UTF-8?'>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <SOAP-ENV:Body>
    <ns1:bind xmlns:ns1="urn:SchedulerServer">
      <SchedulerServer xsi:type="ns1:SchedulerServer" SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
        <SchedulerServer xsi:type="xsd:string">072719153196</SchedulerServer>
        <service xmlns:ns2="http://schemas.xmlsoap.org/soap/encoding/" xsi:type="ns2:Array" ns2:arrayType="ns1:Connection[2]">
          <connection xsi:type="ns1:Connection">
            <id xsi:type="xsd:string">45819154896</id>
            <sourceAddresses xmlns:ns2="http://schemas.xmlsoap.org/soap/encoding/" xsi:type="ns2:Array" ns2:arrayType="xsi:string[3]">
              <sourceAddress xsi:type="xsd:string">194.94.198.201</sourceAddress>
              <sourceAddress xsi:type="xsd:string">194.94.198.202</sourceAddress>
              <sourceAddress xsi:type="xsd:string">194.94.198.203</sourceAddress>
            </sourceAddresses>
            <destinationAddresses xmlns:ns2="http://schemas.xmlsoap.org/soap/encoding/" xsi:type="ns2:Array" ns2:arrayType="xsi:string[2]">
              <destinationAddress xsi:type="xsd:string">194.94.198.127</destinationAddress>
              <destinationAddress xsi:type="xsd:string">194.94.198.128</destinationAddress>
            </destinationAddresses>
          </connection>
          <connection xsi:type="ns1:Connection">
            <id xsi:type="xsd:string">45819154897</id>
            <sourceAddresses xmlns:ns2="http://schemas.xmlsoap.org/soap/encoding/" xsi:type="ns2:Array" ns2:arrayType="xsi:string[2]">
              <sourceAddress xsi:type="xsd:string">194.94.198.201</sourceAddress>
              <sourceAddress xsi:type="xsd:string">194.94.198.202</sourceAddress>
            </sourceAddresses>
            <destinationAddresses xmlns:ns2="http://schemas.xmlsoap.org/soap/encoding/" xsi:type="ns2:Array" ns2:arrayType="xsi:string[1]">
              <destinationAddress xsi:type="xsd:string">194.94.198.318</destinationAddress>
            </destinationAddresses>
          </connection>
        </service>
      </SchedulerServer>
    </ns1:bind>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Listing 10: `bind()` – sample SOAP response

```
<?xml version='1.0' encoding='UTF-8?'>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <SOAP-ENV:Body>
    <ns1:bindResponse xmlns:ns1="urn:SchedulerServer" SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
      <result xsi:type="xsd:boolean">true</result>
    </ns1:bindResponse>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```