CMST 12(1) 47-68 (2006)
DOI:10.12921/cmst.2006.12.01.47-68

COMPUTATIONAL METHODS IN SCIENCE AND TECHNOLOGY **12(1)**, 47-68 (2006)

Programming Grid Applications with Gridge

Juliusz Pukacki*, Michał Kosiedowski, Rafał Mikołajczak, Marcin Adamski, Piotr Grabowski, Michał Jankowski, Mirosław Kupczyk, Cezary Mazurek, Norbert Meyer, Jarek Nabrzyski, Tomasz Piontek, Michael Russell, Maciej Stroiński, Marcin Wolski

Poznań Supercomputing and Networking Center, Poznań, Poland
e-mail: pukacki@man.poznan.pl

Abstract: Not many fully integrated Grid solutions exist on today's market. In this paper we present the PSNC's Grid toolkit, called Gridge, which is fully integrated Grid environment, consisting of tools and Grid services to enable challenging scientific and commercial applications on the Grid.

Key words: Grid computing, Grid services, SOA

1. INTRODUCTION

PSNC is an internationally recognised leader in using, innovating and providing HPC, Grid and network technologies to enable advances in science and engineering. Focusing on data-oriented and computational intensive science and engineering applications, PSNC is considered as an international resource for applied Grid computing research. It has been involved in Grid research from the very beginning of the concept. In 2000, the European Grid Forum (EGrid) was funded at the first kick-off meeting in Poznań. Further, PSNC played an active role in founding the Global Grid Forum (GGF), which was eventually established by merging the EGrid with Grid Asia and US Grid Forum. Since European Commission has realized the importance of new Grid technologies and their deployment on high-speed networks, PSNC researchers have been working on new architecture, design and development of technologies and systems for the Grid. While promoting global adaptation of Grid environments and tools PSNC has created many Grid-enabled applications and services for business and eScience at national and European levels under the scope of many IST projects. Some of them along with their objectives are mentioned below:

- GridLab was one of the biggest European research undertakings in the development of application tools and middleware for Grid environments. It produced a set of application-oriented Grid middleware services and toolkits providing capabilities such as dynamic resource brokering, monitoring, data management, security, information and adaptive services, and more.
- CrossGrid delivered tools and mechanisms for developing and running interactive Grid-enabled applications addressing realistic problems in medicine, environmental protection, flood prediction, and physics analysis.
- HPC-Europa aims to provide advanced computational services to users dispersed across Europe,
- InteliGrid delivers a generic Grid-based integration and web semantic-based interoperability platform for creating and managing global-scale networked Virtual Organizations for virtually partnering SMEs.

Additionally, many national initiatives, supported by the Polish State Committee for Scientific Research, were launched for which PSNC researchers and developers have created useful and ready-to-use eScience platforms and services. Three of them:

- PROGRESS Grid-portal framework for Grid processing and data management,
- Virtual Laboratory, a distributed workgroup environment, with the main task of providing a remote access to the various kinds of rare and expensive scientific laboratory equipment and distributed computational and data resources,
- National Cluster of Linux Systems (CLUSTERIX), a distributed new-generation PC-cluster (or meta-cluster),

were the most prominent ones.

The essence of Grid middleware layer, located and tightly connected with the optical network infrastructure and its services, lies in its ability to provide a scalable, secure, robust and dynamically-configurable advanced communication platform and resource sharing. Therefore, PSNC has created recently an internal open-source software initiative, called *Gridge – The Grid Enterprise Solution*, promoting secure, flexible and adaptable components created and maintained in all aforementioned projects. The Gridge middleware developments at PSNC focus primarily on re-engineering existing middleware func-

tionality achieved in the above mentioned projects. This paper presents the main components of Gridge. The way in which they can be used to support dynamic applications on the Grid is also presented. The rest of the paper is organized as follows: in section 2 we describe Gridge tools in detail. Section 3 presents an example application scenario supported by Gridge. Section 4 shows behind-the-scenes interactions of all the services in the scenario. We conclude with section 5.

2. GRIDGE TOOLKIT

Gridge Toolkit consists of the following tools and services:

- GridSphere Portal Framework (developed within GridLab)
- Grid Service Provider (GSP) (developed within Progress)
- Grid Resource Management System (developed within GridLab)
- Grid Authorization Service (developed within GridLab)
- Grid Mobile Services (developed within GridLab)
- Grid Data Management System (developed within Progress)
- Migrating Desktop (developed mainly within CrossGrid and continued in other projects).
- Grid Monitoring System (Mercure) (developed within GridLab)
- System level checkpointing library

All the pieces are integrated with each other and follow the same interface specification rules, license, quality assurance and testing, distribution *etc*. In the paper the Gridge Toolkit is presented from a programmer's point of view. We also focus on the most important Gridge services and show how they are exposed to the Grid world. Finally, we demonstrate a real application scenario, in which almost all the Gridge services take part.

2.1. User Access

Users can access Grid infrastructure either through portals, mobile devices or command-line clients. Two portal frameworks are available to users: Grid Service Provider and GridSphere Portlet Framework. Access through mobile devices is also available.

1) Grid Service Provider: The user access philosophy in Gridge is drawn around the concept of enabling multiple independent user access applications, such as web portals, standalone applications and mobile user interfaces for the utilization by the users of the Grid infrastructure. In this philosophy each user may execute his Grid work using any of the available user interfaces and may be doing any part of his work using any of these interfaces. Such an approach required a special attention concerning the quick delivery of important Grid data such as application descriptions or job configurations and statuses, and concerning the construction of a single point of entry to many independent and sometimes also heterogeneous Grid environments. This motivation led us to the design of the Grid Service Provider (GSP) module [1]. GSP is a set of high level Grid services whose primary task is to support various types of Grid user interfaces in quick and seamless access to the data and information flowing in from the lower level services. Thanks to the introduction of GSP these data and information can be easily shared between heterogeneous user interfaces without having direct access to the lower level services. For example, it is not necessary to contact a Grid execution engine to check the configuration of a running Grid jobs or to check their status: all this information is stored in the GSP database and can be quickly read and delivered to a user.

GSP contains two high level services: the Job Submission Service and the Application Management Service [2]. The Job Submission Service delivers functions for computing job building, submitting them to the Grid for execution and viewing the results. It allows to create jobs, configure their tasks and set the requirements for Grid environment resources. The Job Submission Service features the Grid resource broker plug-in mechanism which allows it to cooperate with multiple independent Grid infrastructures, thus allowing the users to submit exactly the same job to two different Grid environments. The task of a Grid resource broker plug-in is to communicate with the Grid execution engine in the communication protocol used by that engine and to translate the job structure into the language used by that engine. For example, a plug-in for the Gridge Resource Management System is familiar with GRMS access interface and with the XRSL language used by that service. The Application Management Service manages the Gridge application repository that can also be shared between various independent user interfaces. An application descriptor contains a reference to the application's executable and a set of its available, required or optional arguments, required environment variables as well as input and output files. One executable may be referenced by many applications, and different application configurations are viewed as independent applications. Both GSP services support workflows: the Job Submission Service allows to configure workflow jobs and send them for the execution in the Grid, and the Application Management Service allows to create descriptors for workflow applications and use them as the base for the creation of new job configurations.

2) Grid Portal: In Gridge, the portal access to Grid services is organized with the use of the GridSphere portal framework [3]. Created by the Portals Work Package in the GridLab Project [4], GridSphere leverages the most relevant standards, best-practices and technologies to offer a framework for developing Grid portals. One of the most

exciting standards to gain adoption by the general community is the Portlet Java Specification Request (JSR 168) [5]. The Portlet JSR defines an application programming interface (API) and model for packaging and presenting Web content as *portlets*. Portlets are Java classes that have a clearly defined interface and life cycle. Portlets are hosted by a *portlet container* and can be presented in a Web page in any manner supported by the *portlet container*. The Portlet JSR makes it possible to distribute and share Web applications more easily, creating a means for collaborating on Web portal development on a much larger-scale.

The GridSphere Project [6] has also developed a generic framework for developing Grid portal applications called Grid Portlets [7]. Grid Portlets offers developers a collection of "portlet services" for performing tasks on the Grid. These portlet services can be used to Grid-enable any portlet web application. Grid Portlets provides a collection of simple, easy-to-use, well integrated portlets that showcase the functionality offered in Grid Portlets, including portlets for retrieving credentials, monitoring resources, submitting jobs and managing remote files.

GridSphere can also be used as an environment to run specialized portlets cooperating with the services of the Grid Service Provider and of the Data Management System (DMS). These portlets, developed with the use of specially designed Portlet Framework that allows to run the portlets also in a standalone mode, utilize the high level functionality provided by GSP and DMS to organize more user-friendly access to Grid services. The available portlets that allow to access the functionality of the GSP and DMS services include the "Applications" portlet which allows to manage the Gridge application repository, "My computing Jobs" portlet which allows to create and submit Grid jobs on top of any application available in the repository and "My data" portlet which allows to manage data files stored by the Data Management System. In addition to these core portlets, several specialized application portlets have been developed as examples of user friendly portal interfaces to Grid applications (see Fig. 1). GridSphere is presented in detail in paper of M. Russell *et al*. in this journal (pp. 89-97).

Fig. 1. Specialized application portlet wizard

The main motivation behind the introduction of the Portlet Framework was to create a solution that supports developers of specialized user interfaces to different Grid applications. Thanks to the architecture and technology used by the framework developers of specialized application portlets gain opportunity to create new portlets and enable new Grid applications through easy-to-use job configuration wizards on the portal within several days [8]. The Portlet Framework together with the GridSphere portal container and the Grid Service Provider provide a flexible set of tools to construct web-based Grid access environment. In simple scenarios, involving usage of a limited number of simple applications, a GridSphere installation with the Grid Portlets fulfills the requirements. When the user access involves more sophisticated scenarios with multiple different applications utilized by multiple different user groups, the Grid Service Provider module accessed via the portlets created with the use of the Portlet Framework acts as a high level support for the administrators, developers and users.

3) Mobile access: The typical Grid application incorporates a large amount of data and heavy weight protocols that are used in connection between entities in the Grid. Additionally, processing such huge volumes of data requires much processor power. On the other hand, mobile devices are by nature "limited": they have limited processing power, limited memory, limited network connections. The aforementioned problems led to development of a gateway between mobile client and "heavy-weight" Grid world. The gateway "talks" to Grid services in the name of the mobile applications. It serves only the needed data in a form and size suitable for mobile devices, securing only the most important data. In our approach, it is a web application (the Mobile Command Center, MCC) developed as a Java servlet (also with portlet management interface) integrated with GridSphere framework services (see Fig. 2). The Mobile Command Center is a central point of mobile architecture. All requests sent from the mobile device are served there. If the request requires to call the external Grid service (like Gridge Visualization Service for Mobiles, Gridge Message Box, Gridge Authorization Service or Gridge Resource Management Service), it is translated into an appropriate form (*e.g*. GSI Grid Service gSOAP call) and forwarded to the external service. On the mobile side there is the J2ME MIDP 1.0 enabled device, which is running our Gridge Mobile Client (GMC) Midlet application. On devices that do not provide native J2ME MIDP support it can be used device's custom Java with the ME4SE package, which allows to run J2ME midlets under Java SE or Personal Java environments. The client is a multithreaded Java application, which sends HTTP requests to the server and displays responses in an appropriate way. To avoid authenticating the user with each request, there is a session maintaining (with cookies) implemented in the midlet. All typed data is stored in Record Management System (RMS) record stores to avoid frequent retyping. After login to the server the user can choose

Fig. 2. Gridge Mobile Client overview

which service he/she is going to use, choosing entries from the menu (*e.g.* displaying folders from the Message Box, Showing Visualization Service visualizations or displaying the list of user jobs managed by GRMS). There is a possibility to automatically parse messages for URLs with visualizations – the user does not have to type anything in this case. An application after finishing some tasks can send the information about it *via* the Message Box – the user gets this information as an SMS and using GMC can browse his/her messages, choose the message he/she is interested in, and display the message. If the message contains information about visualizations, the user is notified about it and can display one of them using the client. If the visualization consists of more than one image, the navigation slider is displayed and the user can choose which "frame" of visualization he/she wants to display. Each visualization frame can be cropped and zoomed. The zoom operation can be repeated many times and in such a case the crop operation is always performed on the source image, not the one currently displayed on the device. Such an aproach prevents from data loss (the source images are high resolution: even 1000% zoom on 100 by 100 pixel screen are clear and can be easily read by a user). The cropping procedure is also user-friendly. A resizable image sub-frame is displayed, the user can move it over the picture, resize the frame if needed and request the needed part of the picture with one key press, and the sub-picture is automatically enlarged to the device screen resolution. Another part of the Gridge Mobile Client is connected with the GRMS service. A user can display a list of currently running jobs, start the job or cancel, migrate the existing jobs. There is also a possibility to display job information or history and register for notification messages from the application.

2.2. Resource Management – GRMS

The Gridge Resource Management System (GRMS) is an open source meta-scheduling system, which allows developers to build and deploy resource management systems for large scale distributed computing infratructures. GRMS, based on dynamic resource selection, mapping and advanced scheduling methodology, combined with feedback control architecture, deals with dynamic Grid environment and resource management challenges, *e.g.* load-balancing among clusters, remote job control or file staging support. Therefore, the main goal of GRMS is to manage the whole process of remote job submission to various batch queuing systems, clusters or resources. Finally, GRMS can be considered as a robust system which provides abstraction of the complex Grid infrastructure as well as a toolbox which helps to form and adapts to distributing computing environments.

GRMS (see Fig. 3) has been designed as an independent set of components for resource management processes. It can take an advantage of various low-level core Grid services, such as *e.g.* GRAM, GridFTP and Gridge Monitoring System, as well as various Grid middleware services, *e.g.* Gridge Authorization Service, Gridge Data Management Service and more. All these services working together provide a consistent, adaptive and robust Grid middleware layer which fits dynamically to many different distributing computing infrastructures. The GRMS implementation requires Globus software to be installed on Grid resources, and uses Globus Core Services deployed on resources: GRAM, GridFtp, MDS (optional). GRMS supports Grid Security Infrastructure by providing the GSI-enabled web service interface for all clients, *e.g.* portals or applications, and thus can be integrated with any other middleware Grid environment. One of the main assumptions for GRMS is to perform remote jobs control and management in the way that satisfies Users (Job Owners) and their applications requirements as well as constraints and policies imposed by other stakeholders, *i.e.* resource owners and Grid or Virtual Organization administrators. All users requirements are expressed within XML-based resource specification documents and sent to the GRMS as SOAP requests over GSI transport layer connections. Simultaneously, Resource Administrators (Resource Owners) have full control over resources on which all jobs and operations will be performed by appropriate GRMS setup and installation. Note, that the GRMS together with Core Services reduces operational and integration costs for Administrators by enabling Grid deployment across previously incompatible cluster and resources. Technically speaking GRMS is a persistent service within a Tomcat-/Axis container. It is written completely in Java so it can be

deployed on various platforms. With the GAS, GRMS is able to manage both, job grouping and jobs within collaborative environments according to predefined VO security rules and policies. With the Data Management services from Gridge, GRMS can create and move logical files/catalogs and deal with data intensive experiments. Gridge Monitoring Service can be used by GRMS as an additional information system. Finally, Mobile service can be used to send notifications via SMS/emails about events related to users' jobs and as a gateway for GRMS mobile clients. GRMS is able to store all operations in a database. Based on this information a set of very useful statistics for both end users and administrators can be produced. All the data is also a source for further, more advanced analysis and reporting tools.

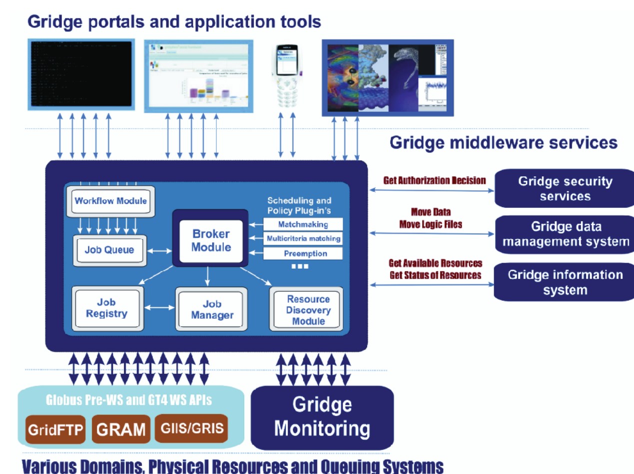


Fig. 3. GRMS architecture

GRMS is composed of the following modules:

- Job Receiver Module – provides GSI enabled web service interface for GRMS. In job submission this module is responsible for job description validation and putting proper job to a queue. For workflow jobs it creates graph representation of tasks and check its correctness.
- Job Queue – stores jobs which are ready for execution. It is prepared for implementation of any queue management strategy and scheduling algorithms.
- Broker Module – it is the heart of GRMS. It steers whole process of job submission: gets jobs from queue, calls Resource Discovery Module to find appropriate resources, evaluates resources to find "the best" one, creates environment for job execution by transferring input data, calls Job Manager Module to monitor status changes of job, after job is finished takes care on transferring output data to location specified by a user. It also provides information about jobs and their history in the system.
- Resource Discovery Module – finds resources that fulfill requirements described in Job Description. Resources are described in an XML document which contains parameters important from the job scheduling point of view. Resource Discovery Module can be configured to use many information sources. It can use *e.g.* Globus MDS, iGrid service, Mercury Monitoring Service, Adaptive Components Service, Testbed Information Service.
- Job Manager Module – responsible for monitoring of status changes of job, and for job control: job canceling, suspending and resuming.
- Job Registry Module – responsible for storing all information about jobs, and make it available for other modules.

An XML based GRMS Job Description (GJD) language was specified to allow users defining the computing jobs and resource requests. For each task there is a section in a job descritpion document describing resource requirements and user preferences used for dynamic resource discovery. Another section defines the application: executable, input and output files required, arguments, environment, *etc.*

One of the most interesting features of GRMS is its ability to deal with jobs defined as a set of tasks with precedence relationships (workflows). With just one call a user can submit the whole computational experiment that consists of many independent application executions. Two ways of expressing the dependencies between tasks are possible. The first one is a direct way, based on the parent-child relationship among tasks. In his case the execution of a child depends on the status change of its parents. The second way of expressing dependencies is associated with a data flow between the tasks. Here a user can specify that an output of one task becomes the input for the other one. What is a beauty here is that a user does not have to specify the exact file locations. However, in such a case it is user's responsibility to define file dependencies correctly. GRMS will reject execution of the job where data dependencies contradict the parent-child relationship. As it was already mentioned the basic way of introducing dependencies between tasks is by defining the parent-child dependencies. A very interesting and novel feature of GRMS which distinguishes it from the other systems is that execution of a child task can be triggered by any status change of a parent task. So, not only a task termination can trigger following executions. This feature is very useful in many scenarios. For instance we can imagine that a user would like to execute some application as soon as the other one starts running – *e.g.* for client server communication. The other example could be the flow of computation that depends on the failure of execution of one of the tasks (failover mechanisms).

GRMS provides programming interfaces (API) to application developers. One of the ways of application integration with GRMS is to use the SAGA API (SAGA is

presented in the paper SAGA: *A Simple API for Grid Applications* on the pages 7-20 of this journal. Actually, GRMS comes with SAGA and GAT adaptors. Here we present the GRMS API as to be used directly from the application. GRMS API and capabilities it provides for end users can be divided into several logical groups according to offered functionality:

- Job submission and control
  This group of functionalities allows to submit and control the whole jobs. Using the submitJob method user can submit to the system a set of dependent tasks constituting one logical job. If the job description is valid (has no syntax and logical errors) the globally unique job identifier is returned that unambiguously identifies the job in the system and can be later used to control it. While submitting the job GRMS supports two phase commit mechanism. In this case the job submitted to the system is not processed until the processing will be approved by commitJob method. Processing of the job can be canceled by cancelJob method or temporary suspended and later resumed (optionally with new job description) by suspendJob and resumeJob methods. Because experiments (jobs) controlled by GRMS can potentially consist of huge amount of time consuming tasks the proper execution of which depends not only on the correct job description, but may be also broken due to some unpredictable events, GRMS allows to restart the job, skipping execution of previously finished tasks invoking recoverJob method. For job submission and then execution of tasks on resources GRMS uses the time-limited user proxy, which can expire during the processing of job causing lost of control on running tasks and making impossible to start new ones. This can happen very often, especially for long running jobs with hard to predict finish time. Addressing the issue GRMS allows to prolong the user proxy for the whole job (refreshJobProxy method).
- Task control
  GRMS allows to control every single task belonging to a job. Execution of every task can be canceled (cancelTask method) or suspended and resumed (suspendTask, resumeTask methods). Processing of every task, irrespective of precedence constraints resulting from dependencies between tasks, can be also postponed until it will be approved by commitment (commitTask method). The migrateTask functionality allows to migrate single, running task to a "better" resource (if such one exists) to improve task performance or system utilization. To be migrateable a task has to be checkpointable. We can distinguish between two cases of job checkpoint supported by GRMS. The first one is the situation when application is checkpointed on demand. This situation is implemented by relatively simple "checkpoint" web service interface, location of which is registered in GRMS or is able in proper way to serve checkpoint command sent by Mercury Service. In both cases the whole process of task migration is relatively simple. The task to be migrated is checkpointed on the resource where it is currently running and then restarted on a new one pointed by the user or chosen by GRMS. GRMS is able to migrate also applications that do not support the aforementioned interfaces. In this case an application has to perform checkpoint procedure periodically saving files to disk and during the migration process it is just killed by GRMS and then the execution is resumed from the state saved in the last checkpoint file. Both described above cases are typical examples of application/user-level checkpointing, requiring from the application developer to implement mechanisms for storing application data to checkpoint files, and assumes that checkpointing procedure is hard coded in the application. The migration process can be performed by GRMS according to a new job description passed as the parameter. If the new job description was not defined, GRMS tries to perform the request based on the job description or the previous migration request. Addressing many advanced scenarios GRMS is able to handle tasks having some time constrains and requirements, like for example specified period of time when the execution of task must be started or the duration of task execution. The extendTaskExecutionTime method allows to prolong the execution time of scheduled tasks.
- Listing jobs according to some criteria
  GRMS returns a list of jobs (identifiers) belonging to the user that invoked the request (getJobsList) or to a specified project (getProjectJobsList). It is possible to query for all jobs or a subset of jobs in given state in the system. The list of jobs can be also limited to some amount of last jobs.
- Listing tasks belonging to the given job according to some criteria
  The getTasksList returns collection of task identifiers belonging to the given job. It is possible to narrow the list down only to tasks in one of specified states.
- Managing tasks
  This subset of GRMS functionality can be divided into two groups. First of them gives possibility to register (registerTaskApplicationAccess), unregister (unregisterTaskApplicationAccess) and query (getTaskApplicationAccess) location of the web service implementing checkpoint interface. The second group of methods concerns dynamic management of files and directories. In a real scenarios very often application doesn't know in advance names of checkpoint files or input files produced by other tasks, because their names can contain timestamps or iteration step of execution. Addressing such cases GRMS allows the application

dynamically to add new files and/or directories (addTaskFileDirs method) or to remove ones specified in job description or dynamically registered in previous calls (removeTaskFileDirs). Additionally it is possible to query for files and directories registered for given task specifying optionally some criteria (getTaskFileDirs method).

- Getting information about jobs
  The getJobInformation method returns complex information concerning particular job. The following information is available: the project to which a job belongs, distinguished name of the owner of the job, current status of the job, time when the job was submitted and finished (if it is known), message describing the cause of last error, list of tasks forming the job, number of tasks and description of the job.
- Getting information about tasks
  The getTaskInformation method returns complex information about the given task including history of its migrations. General information is: type and status of the task, times when the task was submitted and finished (if they are known), time how long the user proxy will be still valid, status of the task processing request, description of last error, length of the full task history. Because every task can be migrated many times and can be executed during its life on different machines GRMS provides the history of each task. Every item of task history contains following information: time when GRMS started processing a task on the given resource, time when the task execution was submitted to the local resource, times when the execution started and finished on local resource (if these times are known), description of task – part of job description concerning the given task, array describing co-allocation of subtasks (needed for mpichg tasks), location of "checkpoint" web service interface registered for the task.
- Getting list of resources that meet user's requirements and criteria GRMS is able list resources that meet user requirements expressed in the job description for a specified task (findResources method).
- Managing notifications
  GRMS provides also support for events notification. Notification mechanisms is very general and designed to allow clients to receive information in asynchronous way. User can register for notifications concerning the whole job (registerJobNotification) or single tasks (registerTaskStatusNotification, registerTaskRequestStatusNotification). The difference except the obvious one is that in case of tasks GRMS is able to send to the registered clients two kinds of notifications: the "status notification" connected with changes concerning a life cycle of the task and the "request notification" related to the performed GRMS request. Jobs have only "status notifications". Currently GRMS is able to send notifications in two ways: using SOAP protocol and writing to a remote file. Registered notifications can be queried according to some criteria (getJobStatusNotifications, getTaskNotifications), unregistered (unregisterJobNotification, unregisterTaskNotification) and detailed information about them can be provided (getJobStatusNotification, getTaskNotification, getTaskStatusNotification, getTaskRequestStatusNotification).
- Auxiliary functionality
  Functionality belonging to this group has no productive character, but can be useful for testing and administrative purposes. GRMS gives the possibility to check the correctness of the job description (testJobDescription method). In the case of incorrectness of description GRMS returns the diagnostic information describing the syntax or logical error. GRMS is able to return user defined description of the service (getServiceDescription) and to list all jobs in given state in the system (getAllJobsList method).

GRMS job description can be divided into several parts describing the way the job should be processed as a whole. Job description starts with general properties characterizing the job in GRMS system. User has to specify a string distinguishing the job from other ones. The name of a job will be used by system as a part of final GRMS job identifier. Optionally it is possible to specify the project, a job belongs to or to specify if the processing of the job needs commitment to be started. The job consists of set of dependent task and job as well as each single task can have notes containing informal and human readable descriptions. Every task forming a job has a set of general properties. It has to have an unique identifier, that distinguishes it from other tasks. Additionally user can specify a type of the task as a persistent or nonpersistent one. The difference is that for persistent tasks GRMS doesn't remove working directories after they finish, what is the default system behavior. Optional "extension" attribute allows to specify in a transparent way that the task has to be started in the working directory of another task, previously submitted and executed as a persistent one. For every task it is possible to specify if it is crucial for the processing of the whole job and if its processing needs a commitment to be started. Next optional section specifies requirements concerning resources. If this section exists, making a ranking of resources, which the task can be mapped to, GRMS takes into consideration only resources that meet these requirements, like minimal number of processors, operating system, type of queuing system and many others. It is possible to specify many sets of resource requirements for each task, describing alternative characteristics of resource that meet user's requirements. Regardless of preferences concerning static properties of resources every task can have additionally hard and soft constraints being used to

rank resources in multi-criteria scheduling process. Main and obligatory part of description of each task is a section concerning executable. For every task location of the executable, which can be url to the physical location of logical identifier, has to be provided. Additionally it is possible to specify if the task is checkpointable or not and all information needed for task execution: arguments, environment variables, input and output files and directories including checkpoint ones and locations of standard input, output and diagnostic streams. It is also possible to express preferences and requirements concerning time of execution, like for example duration of execution, start and end times of a period during which the task must be started, slot within a day when a task must be executed and others. For each task it is possible to specify information about other tasks it depends on and statuses of these tasks that are required to trigger its processing.

2.3. Authorization Service – GAS

The Gridge Authorization Service (GAS) is an authorization system which can be the standard authorization decision point for all components of a Grid system. Security policies for all system components can be stored in GAS. Using these policies GAS can return an authorization decision upon the client request. GAS has been designed in a way that makes it is easy to perform integration with external components and to manage security policies for complex systems (see Fig. 4). Full integration with the Globus Toolkit and many other Grid services makes GAS an attractive solution for Grid envirnoments.

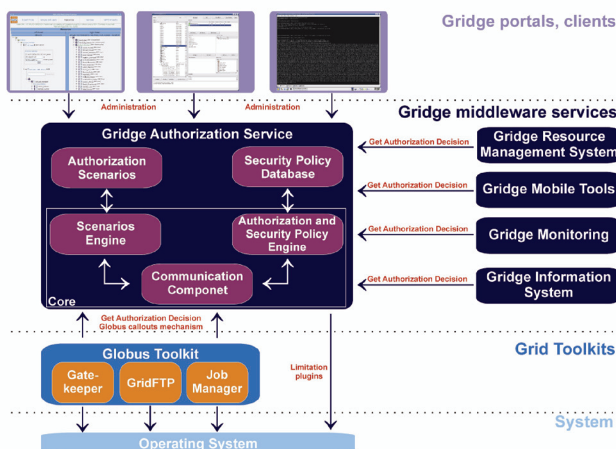


Fig. 4. GAS architecture

As stated above an authorization service can be used for returning an authorization decision upon the user request. The request has to be described by three attributes: user, object and operation. The requester simply asks if the specific user can perform the operation on the specific object. Obviously, the query to an authorization service can be more complex and the answer given by such service can be complicated as well. By using the modular structure of GAS it is easy to write a completely new communication module. The GAS complex data structure can be used to model many abstract and real world objects and security policies for such objects. For example, GAS has been used for managing security policies for many Virtual Organizations, for services (like Gridge Resource Management Service, iGrid, Mobile Services and other) and for abstract objects like communicator conferences or HPC centers in Europe.

The main goal of GAS is to provide a functionality that would be able to fulfill most authorization requirements of Grid computing environments. GAS is designed as a trusted single logical point for defining security policy for complex Grid infrastructures. As flexibility is the key requirement, it is to be able to implement various security scenarios, based on push or pull models, simultaneously.

Secondly, GAS is independent of specific technologies used at lower layers. It should be fully useable in environments based on Grid toolkits as well as other toolkits. The high level of flexibility is achieved mainly through the modular design of GAS and efficient data model, with which one can define many scenarios and objects from the real world. It means that GAS can use many different ways for communication with external components and systems and many security data models and hold security policy on different types of storage systems. These features make GAS attractive for many applications and solutions (not only for those related with Grids). GAS has to be the trusted component of each system in which it is used and brings about that the implementation of GAS was written in ANSI C. This choice makes GAS a very fast and stable component which consumes not much CPU power and little amount of memory.

The main problem of many authorization systems is their management. It is not easy to work with a complex system in a user-friendly way. Based on many experiences and the end user feedback the GAS administration portlet (web application) is provided, which makes management as easy as possible. Flexibility of this solution gives users a full possibility of presenting only these security policies which are important for them. The GAS management is possible in two other ways: by the GUI GTK client and by the command line client.

For Globus Toolkit users, GAS provides a set of plug-ins for Globus components, (for example: gatekeeper and jobmanager plug-in). These plug-ins communicate with GAS in a secure way and can ask GAS about an authorization decision.

2.4. Monitoring Services – Mercury

The Mercury Grid Monitoring System has been developed within the GridLab project. It provides a general and extensible Grid monitoring infrastructure.

Mercury Monitor is designed to satisfy specific requirements of Grid performance monitoring: it provides monitoring data represented as metrics via both pull and push model data access semantics and also supports steering by controls. It supports monitoring of Grid entities such as resources and applications in a generic, extensible and scalable way.

The Mercury Monitoring is designed to satisfy requirements of Grid performance monitoring: it provides monitoring data represented as metrics *via* both pull and push access semantics and also supports steering by controls. It supports monitoring of Grid entities such as resources and applications in a generic, extensible and scalable way. It is implemented in a modular way with emphasis on simplicity, efficiency, portability and low intrusiveness on the monitored system.

The aim of the Mercury Monitoring system is to support the advanced scenarios in Grid environment, such as application steering, self-tuning applications and performance analysis and prediction. To achieve this the general GGF GMA architecture is extended with actuators and controls. Actuators are analogous to sensors in the GGF GMA but instead of gathering information, they implement controls and provide a way to influence the system.

The architecture of Mercury Monitor is based on the Grid Monitoring Architecture (GMA) proposed by Global Grid Forum (GGF), and implemented in a modular way with emphasis on simplicity, efficiency, portability and low intrusiveness on the monitored system.

The input of the monitoring system consists of measurements generated by sensors. Sensors are controlled by producers that can transfer measurements to consumers when requested.

Sensors are controlled by producers that can transfer measurements to consumers when requested. Sensors are implemented as shared objects that are dynamically loaded into the producer at run-time depending on configuration and incoming requests for different measurements.

In Mercury all measurable quantities are represented as metrics. Metrics are defined by a unique name such as host.cpu.user which identifies themetric definition, a list of formal parameters and a data type. By providing actual values for the formal parameters a metric instance can be created representing an entity to be monitored. A measurement corresponding to a metric instance is called metric value.

Metric values contain a time-stamp and the measured data according to the data type of the metric definition. Sensor modules implement the measurement of one or more metrics. Mercury Monitor supports both event-like (*i.e.* an external event is needed to produce a metric value) and continuous metrics (*i.e.* a measurement is possible whenever a consumer requests it such as, the CPU temperature in a host).

Continuous metrics can be made event-like by requesting automatic periodic measurements. In addition to the functionality proposed in the GMA document, Mercury also supports actuators.

Actuators are analogous to sensors but instead of taking measurements of metrics they implement controls that represent interactions with either the monitored entities or the monitoring system itself. In addition to all mentioned features Mercury facilitates steering.

2.5. Mobile User Support

Mobile software development in Gridge (see Fig. 5) is focused on providing a set of applications that would enable communication between Mobile devices, such as cell phones, Personal Digital Assistants (PDA) or laptops and Grid Services on the other side. This class of applications is represented by clients running on mobile devices, mobile gateways acting as a bridge between clients and Grid services as well as additional specialized middleware services for mobile users.

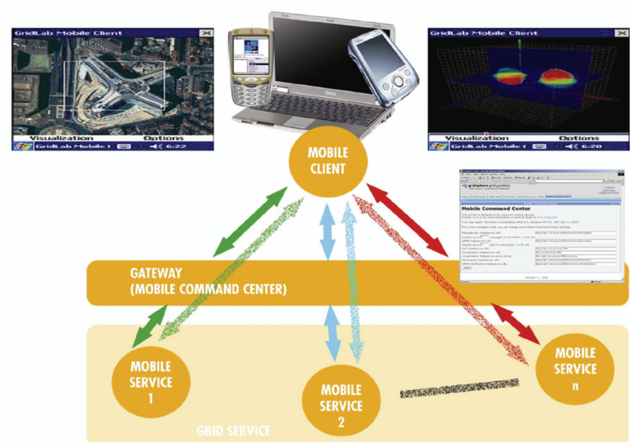


Fig. 5. Mobile Services architecture

The main goal of the services is to make use of small and flexible mobile devices that are increasingly used for web access to various remote resources. The system provides Grid access mechanisms for such devices. This requires adoption of the existing access technologies like portals for low bandwidth connectivity and low level end-user hardware. The mobile nature of such devices also requires flexible session management and data synchronization. The system enhances the scope of present Grid environments to the emerging mobile domain. Utilizing new higher bandwidth mobile interconnects, very useful and previously impossible scenarios of distributed and collaborative computing can be realized. To achieve this and taking into consideration some still existing constraints of mobile devices, the Access for Mobile Users group is developing a set of applications in the client-server model with the J2ME CLDC/MIDP- java client, and portlet server working with GridSphere. This set allow us to manage end

user Grid jobs (steer an application) or view messages and visualizations produced by Grid applications on device such simple as standard mobile phone. The second group of developed services is tightly connected with end user notifications about various events in Grids. Events like: the information about user application is started or finished, the visualization is ready for viewing or waiting for new data, can be send to end users using various notifications way. It can be Email, SMS, MMS, or message of one of Internet Communicators like AIM, Yahoo, ICQ, Jabber *etc*. (including most popular in Poland Gadu-Gadu and Tlen). Mobile services gives also end users possibility to start a conference concerning aforementioned event between users of given virtual organization (including conferences between clients of different communicators).

The unique possibility of giving access to Grid resources for users of relatively weak devices is one of features that distinguish Gridge mobile applications from other Grid systems. Moreover, the used technology, Java 2 Micro Edition – Mobile Information Device Profile (J2ME-MIDP) applications (midlets) on the client side allows to develop flexible, possibly off-line working programs that may be used on a wide range of devices supporting J2ME. Using the MIDP compliant device internal repository for storing data, gives the user possibility to use it later in off-line state and prepare the data, to be sent in on-line state. The Mobile Command Center (MCC) that acts as a gateway between mobile client and Grid services is developed in Java as a GridSphere portlet (see Gridsphere.org) with separate "mobile" context. MCC automatically grabs the device profile (like device class, screen size, color depth, etc), this information is used during forwarding the request from mobile device to Grid services (mainly GSI-enabled Web Services like Gridge MessageBox, Visualization Service for Mobiles or Gridge Resource Management System). Services that can be accessed from mobile device using MCC belong to two groups: the first group consists of Grid services that were adopted to use with mobile devices, the second group are services developed for use only with mobile devices. The Visualization Service for Mobiles belong to second group and is used to view the application output in form of visualization prepared exactly according to the User's device capabilities. The advantage in this case is as follows: the large amount of data is not sent via weak GPRS connections to the device that cannot store it in the memory and cannot display it correctly. First group of services consists of Gridge Resource Management System and Notification and Messenger Service. The first service can be used in 'Collaborative scenario' – the user can steer the application (even not being an owner) from mobile device. He/she can get the jobs list, migrate, resume, suspend, cancel, edit, view history and submit new job on the basis of edited/modified description of already finished jobs. Using GRMS together with Notification service the user can register for user notifications related to the running jobs. In this way the user is notified about important events occurring in the Grid (like jobs status changes, application output availability). These notifications can be send as Email, SMS and Internet Communicator (AIM, Yahoo etc) messages to the user. Using the Messenger Service it is possible also to make a conference between users of Virtual Organization defined in Gridge Authorization Service even if they use different communicators.

2.6. Data Management

Data storage, management and access in Gridge environment is supported by the Gridge Data Management Suite (DMS). This suite composed of several specialized components allows to build a distributed system of services capable of delivering mechanisms for seamless management of large amount of data. This distributed system is based on the pattern of autonomic agents using the accessible network infrastructure for mutual communication. From the external applications point of view DMS is a virtual file system keeping the data organized in a tree structure. The main units of this structure are metadirectories, which enable to put a hierarchy over other objects and metafiles. Metafiles represent a logical view of computational data regardless of their physical storage location.

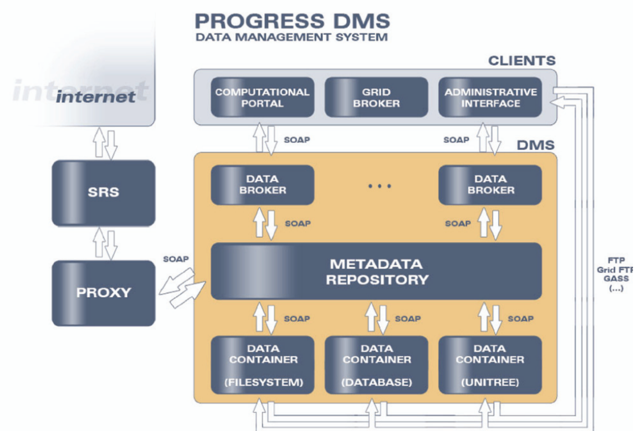


Fig. 6. Gridge Data Management System architecture

As shown in Fig. 6 the Data Management System consists of three logical layers: the Data Broker, which serves as the access interface to the DMS system and implement the brokering of storage resources, the Metadata Repository that keeps information about the data managed by the system, and the Data Container, which is responsible for the physical storage of data. In addition, DMS contains modules which extend its functionality to fulfill the enterprise requirements. These include the fully functional web-based administrator interface and a Proxy to external scientific databases. The Proxy provides a SOAP interface

to the external databases, such as for example those provided by SRS (Sequence Retrieval System) [9].

The Data Broker is designed as an access point to the data resources and data management services. A simple API of the Data Broker allows to easily access the functionality of the services and the stored data. The Data Broker acts as a mediator in the flow of all requests coming from external services, analyzes them and eventually passes to the relevant module. The DMS architecture assumes that multiple instances of the Data Broker can be deployed in the same environment, thus increasing the efficiency of data access from various points in the global Grid environment structure.

The Metadata Repository is the central element of the Gridge distributed data management solution. It is responsible for all metadata operations as well as their storage and maintenance. It manages metadata connected with the data files, their physical locations and transfer protocols that could be used to obtain them, with the access rights to the stored data and with the metadescriptions of the file contents. Currently each DMS installation must contain a single instance of the Metadata Repository, which acts as a central repository of the critical information about the metacatalogue structure, user data and security policy for the whole DMS installation.

The Data Container is a service specialized towards the management of physical data locations on the storage resources. The Data Container API is designed in a way to allow easy construction and participation in the distributed data management environment of storage containers for different storage environments. The Data Containers currently available in the DMS suite include a generic file system Data Container, a relational database Data Container and a tape archiver Data Container. The data stored on the various storage resources can be accessed with one of the many available protocols including such as GASS, FTP and GridFTP.

The Proxy modules are services that join the functionality of the Metadata Repository allowing to list the available databanks, list their content, read the attached metadata attributes and to build and execute queries, and of the Data Container to provide the data using the selected data transfer protocol. Such Proxy container are highly customized towards the specific platform they are working with to allow building complex queries and executing operations on the found entries.

2.7 Accounting – VUS

Identification of users in any system is necessary for accounting and security reasons, *e.g.* in order to charge for used resources and tracing unfair behavior. On the Grid level, the user is uniquely identified by subject of his proxy certificate (so called Distinguished Name – DN). The proxy may also contain some additional information related to the identification, like *e. g.* name of a Virtual Organization on behalf which the user acts. On the other hand, on the operating system level, the user is identified by user account, on which the processes performing user requests are run. Thus we face problem of mapping global user identity (DN) to a local identity (account). The simplest solution is 1-1 mapping, which means the user must have a "personal" account on each node in the Grid (this solution is implemented by Globus gridmap file). This is not scalable and hard to manage in case of bigger systems for obvious reasons. Another simple approach is *n*-1 mapping, where many users may be mapped to the same account. This is usually not sufficient, even if only users of the same organization are mapped to the same account. The mentioned accounting and security requirements are not fulfilled and moreover, problem of unwanted interference of different users' jobs occur.

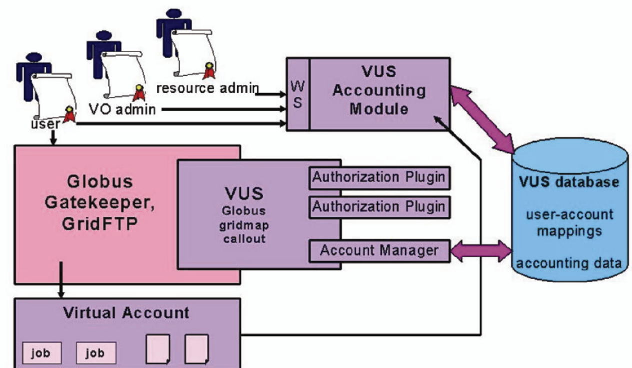


Fig. 7. Virtual User System architecture

The Virtual User System – VUS [10] addresses the problems mentioned in the above paragraph. VUS is an extension of Globus GRAM (gridmap callout) and allows running jobs without having a "personal" user account on a node (see Fig. 7). First, the user is authorized by querying set of Authorization Plugins. The example plugins are gridmapfile (allows for backward compatibility with standard Globus mechanism) and GAS (allows for integration of VUS and GAS). The next step is selection of local account. The "personal" accounts are replaced by "virtual" ones, that are mapped to users only for time needed to fully process a job. The Account Manager assures that only one user is mapped to a particular account at any given time. The history of user-account mappings is stored in a database, so that accounting and tracking user activities are possible.

The local VUS database was designed to store both standard and non standard accounting data types. The standard accounting may be periodically gathered from the local accounting (operating system or local scheduling system level) and merged with global user identity by Accounting Module scripts. Then, the accounting may be published via webservice interface.

2.8 System level checkpointing

Checkpointing provides a service that allows the system administrator or resource management system to store the application state image at any point of computation. The application state image should allow restarting the computation from the point defined by the content of the image. Checkpointing is a desired technology and PSNC has already been involved in the research and development activities in that field since year 2000. By now PSNC has developed two checkpointing packages for Solaris OS and one package for SGI Altix systems running under the SGI ProPack environment:

- **psncLibCkpt** is a user-level library that provides checkpointing functionality for Solaris 8 OS. The core part of the psncLibCkpt is based on the libCkpt library.The most important novelty of the psncLibCkpt is the ability to checkpoint and restart multi-process programs that utilize System V IPC objects to mutual communication and synchronization. It is our first product in which we have introduced the virtualization of identifiers and keys related to processes and System V IPC. Thanks to that, when the program is recovered, it is cheated that the identifiers have not changed (even though due to technological reasons, it is very likely that they have). To utilize the psncLibCkpt library, the program has to be recompiled against the library and the only modification of source codes encompasses replacing with programs written in the name of the main() function with the ckpt_target() name. The library is designed to be used n the C language. No special installation or deployment activities are required so the library can be used even by any regular, not privileged user. The package has been developed as part of the PROGRESS project.
- **psncC/R** is our first kernel-level (system-level) checkpointing package. The product is aimed at Solaris 8 and 9 OS running on UltraSparc CPU. The main advantage of the kernel-level approach is full transparency for programs that are to be checkpointable and independent of the programming language that was used to write these programs. From the end-user's point of view, the utilizing of kernel-level checkpointing package is really convenient and simple but requires some deployment activates that have to be done by the system administrator. Similarly to psncLibCkpt, that package has been developed within the PROGRESS project.
- **Altix C/R** is a kernel-level checkpointing package designed for Altix systems equipped with IA64 processors and running under the ProPack environment (a Linux-based environment prepared by SGI). Currently we have versions of our package that works with ProPack based on linux kernel 2.4 as well as with a more recent ProPack that is based on linux kernel 2.6. The package is characterized by all features typical for kernel-level approach. It is easy to use, there is no assumption on the availability of source codes or the programming language that was used to write the programs that are to be checkpointed. The package has to be deployed by the system administrator. The package allows to do checkpoints of multi-process programs that communicate through System V IPC objects. Additionally, the idea of virtualization of some system global keys and identifiers has been employed in that product as well (advantages of such virtualization are the same as in case of the psncLibCkpt library). The package has been developed as part of the SGIGrid project but we intend to further extend the functionality of that package also beyond the SGIGrid project. Currently we are making every effort to add support for programs that use threads and 'local' sockets. Such features would allow us to prepare the package with the capability of doing checkpoints of some MPI programs.

Contemporary Grid environments are featured by an increasingly growing virtualization and distribution of resources. Such situations impose greater demands on load-balancing and fault-tolerant capabilities. The checkpoint-restart mechanism seems to be the most intuitive tool that can fulfill the specific requirements. Unluckily the Grid environments suffer from the lack of a well-defined interface to the existing and future checkpointing packages nowadays. Therefore the aim in CoreGrid project is to define the high-level checkpoint-restart Grid Service and to locate it among other Grid Services. We defined the Grid Checkpointing Architecture that encompass both, the abstract model of that service and the lower layer interface that allows the service to cooperate with the diverse existing and future checkpoint-restart tools.

3. APPLICATION SCENARIO USING GRIDGE TECHNOLOGY

Using the Gridge, application developer has an opportunity to build complex, advanced Grid application scenarios. In this section, a master-slave task farming scenario using some Gridge services directly from a user application is presented. Since it is not possible to show all the features of the Gridge in one scenario we present most important API calls to services. In order to simplify the presentation of the scenario we neglect the nature of computation performed by the application.

General idea of the scenario is to sketch out the standard use-cases of work with Gridge services. A standard use case consists of such operations as submitting a job, staging the job files, starting the job on a resource or set of resources, controlling the job, migrating it if necessary, *etc*. There are several methods of application submission using Gridge. An application developer can

choose simple command line clients for testing, but a user could prefer GUI such as portal based access.

When the application is started, it can communicate with Gridge services in order to request a variety of actions to be performed on user's behalf. All the services act on behalf of the application and Grid user. Of course the applications still compete for resources, but the whole process of resource management is managed in a configurable way by Gridge services according to rules and policies defined by a Grid administrator.

In the proposed master-slave scenario, the job is submitted to Grid, using GRMS metascheduler. GRMS is responsible for choosing the best resource for the job and for remote execution of application. Since GRMS is equipped with the workflow engine it can handle jobs that consist of many tasks (separate applications). In the example described below the job consists of two applications – Master and Slave. Slave is launched as soon as Master is running on one of Grid resources. Master controls the execution of the experiment in a few ways:

- Monitoring progress of the Slave application
- Migrating the Slave application due to decrease of application performance on current resource
- Spawning additional jobs based on some internal indicators

To serve the application calls, not only application–service communication takes place, but there are also a lot of interactions between services. For instance each usage of remote service function is authorized by the GAS authorization service.

3.1. Job submission

The first step of job submission to the Grid, is to describe an application in a way readable by GRMS. Job Description accepted by GRMS is an XML document which can be constructed by user, or automatically generated by portal according to XML schema. The description of the job for the presented scenario, could look like example below:

```
<?xml version="1.0" encoding="UTF-8"?>
<grmsJob appid="MYJOB">
  <task taskid="MASTER">
    <resource>
      <hostname>host1.man.poznan.pl</hostname>
    </resource>
    <executable type="single">
      <execfile name="master">
        <url>file:////bin/master</url>
      </execfile>
      <arguments>
        <value>--xf</value>
        <value>--verbose</value>
        <file name="parameters" type="in">
          <logicalId>master_param1</logicalId>
        </file>
      </arguments>
      <environment>
        <variable name="SLAVE_ID">SLAVE</variable>
      </environment>
    </executable>
  </task>
  <task taskid="SLAVE" commitWait="true">
    <resource>
      <applications>
        <application>Slave</application>
      </applications>
    </resource>
    <executable type="single"
               checkpointable="true">
      <execfile name="slave">
        <url>file:////bin/slave</url>
      </execfile>
      <arguments>
        <value>2</value>
        <value>25</value>
        <file name="input1" type="in">
          <logicalId>input_from_slave1</logicalId>
        </file>
        <file name="output1" type="out">
          <logicalId>output_from_slave1</logicalId>
        </file>
      </arguments>
    </executable>
    <workflow>
      <parent triggerState="RUNNING">MASTER
      </parent>
    </workflow>
  </task>
</grmsJob>
```

There are two tasks in the job description: MASTER and SLAVE. In a resource requirement section of Master, host name is specified directly, but for Slave, the machine to execute an application will be chosen from the list of resources that have the specified application installed locally (dynamic resource discovery). Executable description contains information about location of file, arguments of the execution, input files required and generated output. Workflow section in the SLAVE task, denotes that Slave has one parent (MASTER task) and will be executed as soon, as the Master passes to the RUNNING state. Of course it is possible to define more than two tasks and with very complex precedence constraints – everything is up to the application developer. Tasks can communicate with each other or can be entirely independent, for instance one can define two independent pairs of Master and Slave in one job description. There is only one condition: Maser has to know the identifier of Slave task, but that requirement is very

simple to meet (*e.g.* using environment variable). Detailed information about rules of job description construction can be found in GRMS User's Guide [11].

1) Command-line client: The simplest way to submit the job to the Grid and to control its execution is to use simple comand-line client (see Fig. 8) offering access to GRMS functionality from console. Detailed information concerning installation, configuration and usage of GRMS comand-line client can be found in GRMS Admin's Guide [12]. Having the client installed and configured properly user has to create proxy invoking grid-proxy-init command from Globus toolkit and then is able to submit job simply typing following command:

```
./ws_client.sh submit_job <jobDescFile>
```

replacing <jobDescriptionFile> with path to the file containing description of a job to be submitted. If the submission process succeeded, GRMS returns the job identifier.

```
[piontek@druid bin]$ ./ws_client.sh submit_job ../gridge/master_slave.xml
– Your DN: /C=PL/0=GRID/0=PSNC/CN=Tomasz Piontek
– Service URL: https://druid-bis.man.poznan.pl:8442/axis/services/grms
– Job submitted successfully, jobId=1085556664951_MYJOB-1620
```

Fig. 8. GRMS job submission with command-line client

2) Mobile client: Another possibility of launching the application, is using Gridge Mobile Client from Gridge

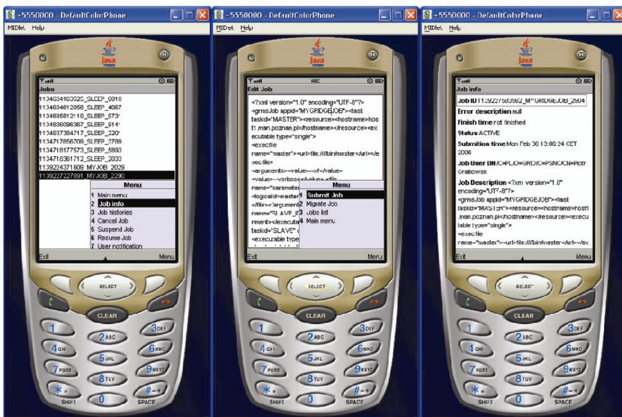


Fig. 9. GRMS job submission with Gridge Mobile Client

services and tools (see Fig. 9). Detailed information about installing and using GMC can be found in Gridge Mobile Client User's Guide [13]. As the first step after launching the midlet, the user should login to Mobile Command Center gateway, which takes care of forwarding user's command to appropriate services. After successful login the user can open the last used GRMS job description file, which is stored in device persistent memory or load the list of previously submitted jobs from the GRMS service and use the job description of one of them. It is also possible to create a new job description, however taking into consideration mobile device typing limitations it is better to load it in one of aforementioned way and alter only important parts of the file. After preparing the job description the user can submit the job using "Submit" command from the menu. The job is submitted and the information about current status of the job is displayed on mobile device screen.

3.2. Application runtime

When the job is submitted to GRMS, metascheduler is taking care of the process of the applications execution, in the proper place, and in the correct order. At first, application of task MASTER is submitted, and as soon as it starts running on the selected machine, Slave is executed. The main goal of the Master is to control the application flow – of course it can have computational code too. After activation it calls GRMS to register for receiving status changes of Slave application. To receive notifications the application can implement appropriate web service interface, or read file accessed by GRMS via http/https interface of GASS server. It should be described by application in the register call using notification parameter:

```
NotificationId registerTaskStatusNotif(
        JobIdentifier jobId,
        TaskIdentifier taskId,
        TaskStatusNotif notification);
```

TaskStatusNotif type describes how GRMS should send required notifications to client – type of notification interface, address of client's interface, message format. Notifications about status changes can also be sent to a person responsible for application submission. In that case, it is realized using SMS text messages, e-mails or internet communicator messages. To register application owner for such notifications, two calls to different services are needed. First the Notification Service has to be registered in GRMS as a destination point for notification events concerning changes of statuses of the given task. Then, knowing the newly created GRMS notification identifier, it can be called the Notification service (one of Gridge Mobile Services). Using this call the service can set the desired message format and sending way. If the call is not done the service assumes that default values should be used. Default value for message format is stored in Notification service properties file. The default notification is stored in per user manner in Gridge Message Box user profile service. Each message can be sent in one or more notification way and stored inside Message Box for further retrieval with Gridge Mobile Client application or Message Box administration portlet. Aforementioned call to Notification service is:

```
void registerNotification(
    NotificationRequest request);
```

NotificationRequest type contains information needed for proper identifying and sending the notification message to the user.

In the meantime, as soon as the Master moved to RUNNING state, execution of the Slave was triggered. But application developer, specified in the job description that task should be suspended until GRMS receives ‘task commit’ call from the client (commitWait attribute of the task tag). That mechanism is very useful to synchronize two tasks – prevent from loosing notifications about status changes. So in the next step, Master calls GRMS to release Slave:

```
void commitTask(
    JobIdentifier jobId,
    TaskIdentifier taskId);
```

The Slave application is considered to do an actual computation. But it is also instrumented with some code, responsible for communication with the middleware services. It will show a dynamic nature of application behavior in Grid environment built with the Gridge services and advantages of that solution. Master is going to manage computation made by Slave but there is no direct communication between them – all steering goes through services. To make it possible the Slave registers two metrics in the Mercury Monitoring Service. First one represents performance – how fast the computations are on the chosen machine, which can vary mainly with changing load of a CPU, but also with many other factors. Second metric, called “progress”, will be used for presenting current state of computations made by the Slave. In order to be able to publish metrics the Slave has to connect to Local Monitor and register itself as a producer using the following method:

```
int prod_app_start(
    const char *prog_name,
    int tid,
    const char *job_id);
```

providing additional information like: name of the application process, thread identifier in case of multithreaded application, job identifier (used only if the application sensor is configured to accept the job id sent by the application). Progress and performance metrics are registered using the same call:

```
int prod_app_register_metric_simple(
    const prod_app_metric_desc *desc);
```

The desc structure describes the control that the application provides. This structure contains information about the name of the metric, its data type and measurement type. Because in opposite to the progress metric, the performance one has continuous character this metric must additionally define the method implementing the sensor with its parameters. The sensor method sends the buffer containing value of calculated metric using method:

```
int prod_app_send_result(
    prod_app_cookie *cookie,
    mon_buffer *value,
    const struct timeval *timestamp);
```

where cookie is an opaque value identifying information and control execution requests, value and timestamp have intuitive meanings. The values of progress metric are sent to the Master using method:

```
int prod_app_send_event(
    const char *metric,
    const mon_arg_list *args,
    mon_buffer *value,
    const struct timeval *timestamp);
```

where metric is the name of the metric, args – the metric arguments, value – buffer containing the value to send and timestamp – timestamp when the event was generated.

For the application executed on machine by GRMS, working directory is created dynamically. After its termination, in default case, the working directory is removed. Because of that, it is important to define in job description, location where the output data (files or whole directories) should be transfered. GRMS is able to handle transferring output to specified destination, expressed as a physical URLs (GridFtp) or a logical location in Data Management System. But not always output data file names are known *a priori*, before job execution. To resolve that problem it is possible to register in GRMS information about additional files (or directories) that will be generated by application. The call to GRMS looks as follows:

```
void addTaskFileDirs(
    JobIdentifier jobId,
    TaskIdentifier taskId,
    FileDir[] items,
    boolean overwrite);
```

Type FileDir describes file or directory: name, url or logical identifier, permissions *etc*. It is obvious that GRMS provides also API calls for viewing information about registered data, and for entries removal. If additional output is going to be stored in Data Management System, location of registered output is expressed as logical identifier. That identifier should be received from DMS using following call:

```
NodeElement addUserDirectory(
        String name,
```

```
        int currentDirID);
FileElement addUserFile(
        String name,
        int currentDirID);
```

These methods create in Data Management System suitably logical directory and logical file with given name and place it in specified currentDirID directory. Both method returns structure containing among other things the numerical identifier of new created entry.

It assures reservation of this unique name for generated file or directory. In proper time GRMS will receive from DMS location where data should be transfered using following call:

```
URI makeUserFileLocation(
    int fileID,
    long fileSize,
    long timePeriod,
    boolean isExactSize,
    String protocol);
```

specifying logical identifier of file, which the location is created for, and information about size of the file and required protocol that will be used to transfer data. If the logical file has already registered location, it can be obtained from DMS invoking following method:

```
URI getUserFileLocation(
    int fileID,
    long timeLock,
    String protocol);
```

that returns location of the file identified by given identifier and accessible by specified protocol. DMS guaranties that the file under the returned location will be accessible for timeLock time.

The last action the Slave can do in its initialization phase is registering interface for application level checkpoint. As it was mentioned before, application can be checkpointed (on application level) in a few different ways. In the most simple case application is periodically storing checkpoint files, and process of checkpointing reduces to killing application process. The alternative way is to instrument application with web service interface that can be used by external service to invoke the checkpoint procedure. When application receives such call it can store an appropriate data and then exit. To register web service interface the following GRMS API call should be used:

```
void registerTaskApplicationAccess(
    JobIdentifier jobId,
    TaskIdentifier taskId,
    TaskApplicationAccess appAccess);
```

In TaskApplicationAccess type argument, address of interface is provided.

It is not allways possible to call application using web service interface. for instance because of firewalls, or in case when the application is running on a cluster node with no public IP address. In a such cases the only way for checkpointing is to use Mercury Monitoring System. For that purpose the application should be instrumented with ability to react in a proper way on checkpoint control sent by Monitoring System. The Mercury API provides two functionally equivalent methods for application developers:

```
int prod_app_register_ctrl_simple(
    const prod_app_ctrl_desc *desc);
int prod_app_register_ctrl(
    mon_metric_def *def,
    prod_app_check check,
    prod_app_sample execute);
```

Both of them can be used to register a new control together with the callback function to activate when the control is invoked. Description of the control to be registered contains information about the name of the control, its type, function to call for checking control parameters, number of these parameters, their types and the control execution function. Basing on this general mechanism it is possible to checkpoint application registering a predefined checkpoint control and implementing inside the callback function the application level checkpointing. The checkpoint request can be sent invoking general method:

```
int monp_cmd_execute_s(
    monp_conn *conn,
    const char *name,
    mon_arg_list *args,
    uint32_t *metric_id);
```

that specifies the name of the control and its arguments.

After all initialization procedures are done, Master's role is to monitor metrics registered by the Slave, and react on its changes, while the Slave is doing actual computation. To be able to do aforementioned monitoring Master has to connect to appropriate producer using the following method:

```
monp_conn *client_connect(
        const char *url,
        const char *auth_meth,
        char **wrappers, int timeout);
```

and providing needed parameters like: location of the producer, authorization method, connection timeout. Being connected Master can query for continuous performance metric invoking method:

```
int monp_cmd_query_s(
    monp_conn *conn,
    const char *name,
    mon_arg_list *args,
    uint32_t *metric_id);
```

and subscribe for progress events invoking folowing sequence of calls:

```
int monp_cmd_collect_s(
    monp_conn *conn,
    const char *name,
    mon_arg_list *args,
    uint32_t *metric_id);
int monp_cmd_subscribe_s(
    monp_conn *conn,
    uint32_t metric_id,
    uint32_t connection);
```

The query method is an optimization for fast information retrieval. It is equivalent to a sequence of COLLECT and GET commands followed by STOP one sent to Monitoring system. Due to its nature it can only be used with continuously measurable metrics. The COLLECT command instructs the monitoring system to create a metric instance with the given parameters and the GET one to send value of metric to the consummer preceding this by new measurement in case of continuous metric. The subscribe method instructs Monitoring system that values of this metric shouldn't be buffered but automatically sent to the specified channel. Regardless of the metric type it is needed to invoke the method:

```
int monp_metric_wait(
    monp_conn *conn,
    uint32_t metric_id,
    monp_metric_value **mv);
```

that waits until a value for the metric identified by metricid arrives and returns it in structure representing a metric value. At the end on its work and during the migration of Slave process Master invokes the following method:

```
int monp_cmd_stop_s(
    monp_conn *conn,
    uint32_t metric_id,
    uint32_t connection);
```

to inform the monitoring system that it is no longer interested in monitoring the given metric and no more metric values for this identifier should be sent to the specified channel. Detailed information concerning Monitoring System can be found in "Adaptive Grid Monitoring Architecture Prototype" documentation [14].

3.3. Checkpointing and migration

As it was mentioned before Master there are two metrics being monitored by Master. One of them represents current performance of Slave application on given machine. While it drops below defined threshold Master decides to move application to less loaded machine. Service responsible for migration process is GRMS, it can be invoked using following API call:

```
void migrateTask(
    JobIdentifier jobId,
    TaskIdentifier taskId,
    JobDescription jobDescription);
```

It is not mandatory to provide a new job description – jobDescription argument is optional. It should be provided to change the way the application will be executed after migration. In the most simple use-case the Master could add some executable arguments that are needed after migration – for instance a parameter that indicates that the migration took place. After receiving migration call the GRMS is trying to checkpoint the Slave application. There are two types of checkpointing available in the Gridge environment: application and system level.

In application level checkpointing application itself is responsible for providing such capability. After receiving the migrate call, the role of the GRMS is to checkpoint the application, find new better resource, transfer all needed data there and resubmit it. As it was already mentioned for the simplest case GRMS just kills the application process and assumes that checkpoint data were generated periodically. If attribute checkpoint of tag task is set to 'true' in job description, the GRMS is trying to use one of checkpointing interfaces, either web service interface of application or appropriate interface of Mercury. While Slave receives checkpoint call it can perform any required actions including registering additional output data and checkpoint files, and finally terminates.

For system level checkpointing GRMS is using the Checkpointing Service that exploits libraries developed for some operating systems as described in section 2.8

3.4. Application termination

All the time, Master application is aware of state of Slave, thanks to notifications sent by GRMS. So it can suspend metrics monitoring during migration and resumes it as soon as application is active again. I was not described so far how the second registered metrics is used. The purpose of it, is to trigger a spawning a new job by the Master. Based on internal algorithms the Master is able to decide that it is required to run new instance of application, doing some independent computation, to increase overall performance. To do so the Master constructs new job description and submits it to GRMS:

```
JobIdentifier submitJob(
        JobDescription jobDesc);
```

Job description can contain one or more tasks describing Slave application, and treated similar way as introduced for Slave so far.

The last step of the scenario is connected with termination of the applications. Before the Slave finishes execution, it has to unregister metrics and checkpoint control from Monitoring using following methods:

```
void prod_app_unregister_metric(
    const char *name);
void prod_app_unregister_ctrl(
    const char *name);
```

The only parameter of both methods is the name of the metric or control to be unregistered.

Then, it can also register with GRMS any additional output generated that was not defined in job description or wasn't registered so far.

When the Slave terminates, GRMS takes care about transferring output data to specified locations and clearing application's workspace. Master can finish its execution when there is no Slave application running. But before exiting it can do some cleaning: unregister notifications about Slave status changes from GRMS:

```
void unregisterTaskNotification(
    JobIdentifier jobId,
    TaskIdentifier taskId,
    NotificationId notificationId);
```

and from Mobile Services (namely from Notification service):

```
void unregisterNotification(
    Notification notification);
```

Notification type contains information needed for proper identifying the notification to be unregistered from Notification service.

If one of the slave results are visualizations the application can enable this output for mobile device users. To do this slave would call the Message Box service from Gridge Mobile Services. Calls to do are:

```
Message createMessage(
    User user,
    Folder folder,
    String messageTitle,
    String messageText,
    MessageAddressData messageTo,
    MessageAddressData messageFrom,
    MessageAddressData messageCC,
    MessageAddressData messageBCC,
    int iFlag,
    Calendar timestamp);

int saveMessage(User user, Message message)
```

User type contains information needed for proper identifying the message box user to create the message for. Folder type contains information needed for proper identifying the message box folder to create the message in. MessageAddressData type contains information needed for proper setting the message address fields. Since this moment the user can read the message with Gridge Mobile Client and if the message contains any link pointing to displayable visualization, the user can view this visualization prepared by Visualization Service for Mobiles exactly for the user device capabilities.

4. BEHIND THE SCENES – SERVICES INTERACTIONS

Presented in section 3 scenario was focused on describing interaction between application and Gridge services. There is a lot of activity taking place among services itself, as a reaction on application calls. First of all, GAS is heavily used as a central point for authorization decision. So when any remote interface of the service is invoked, it checks in GAS if given user is authorized to call that method of the service. To realize job submission call, GRMS calls Information Service (for instance Globus MDS) to search for potential resources to use. Information Service can obtain dynamic part of machine parameters from Mercury Monitoring. After choosing a machine, GRMS uses remote interface of Globus for job submission. Before application actually starts, Virtual User Account System is used for mapping Grid user to local account on given host. In case the job is started from Mobile device with Gridge Mobile Client as the first step the client sends request to login to Mobile Command Center gateway, which forwards this request to GridSphere login service. Depending on the login result, the user can or cannot proceed to GRMS job submission part of mobile client. Assuming the user is logged in, he/she can submit the job using submit command, which is forwarded from gateway to GRMS service. User credentials needed for job submission are obtained by gateway from Gridsphere services. To register for user notification the application calls GRMS service and Notifications Service. When the specified event occurs GRMS is calling the Notification Service to notify the application owner, or group of users with e-mail, SMS or internet communicators messages. If the message should be sent as instant internet communicator message it is forwarded to Messenger Service, which actually establishes a conference between all interested users. All notification messages are stored in Message Box for further retrieval *via* Gridge Mobile Client – so the Email or SMS message, which was send to the user can be simultaneously viewed from mobile device.

After the application terminates GRMS transfers to the output the generated data. It calls Data Management System to provide physical location for given logical file identifier, and then launches the data transfer from working directory of application to some data storage system. If one of the Slave's application results are visualizations, the application can enable this output for mobile device users. To do this, the application calls the Message Box service. Since that moment the user can read the message with Gridge Mobile Client and if the message contains any link

pointing to displayable visualization, the user can view this visualization prepared by Visualization Service for Mobiles (VSfM) exactly for the user device capabilities.

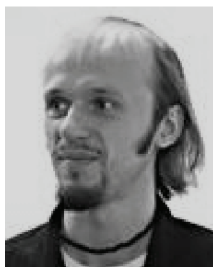
5. CONCLUSIONS

In this paper we have presented the Grid Toolkit called Gridge, developed by Poznań Supercomputing and Networking Center. The way applications can use Gridge is presented using a simple, yet advanced task farming scenario. Gridge can be used as a whole, including all the services and tools, or a user may choose just these services and tools that are important for specific scenario. Gridge provides a flexible, secure and robust Grid infrastructure. It is currently being used in many Grid infrastructures, including Clusterix (http://www.clusterix.pcz.pl), VLab (http://vlab.psnc.pl), InteliGrid (http://www.inteligrid.com), ACGT (http://acgt.ercim.org) and others.

References

[1] GRMS Admin's Guide, http://gridlab.org/Resources/Deliverables/D9.6.Admins_Guide.pdf
[2] GRMS User's Guide, http://gridlab.org/Resources/Deliverables/D9.6.Users_Guide.pdf
[3] Mobile Client User's Guide, http://gridlab.org/Resources/Deliverables/D12.5.userguide.pdf
[4] Adaptive GRID Monitoring Architecture Prototype, http://www.gridlab.org/Resources/Deliverables/D11.6.pdf
[5] M. Jankowski, P. Wolniewicz and N. Meyer, *Virtual User System for Globus based grids*, Cracow '04 Grid Workshop Proceedings, December 2004.
[6] GridSphere Project, http://www.gridsphere.org/
[7] GridLab: A Grid Application Toolkit and Testbed, http://www.gridlab.org/
[8] Java Community Process: JSR 168 Portlet Specification. Project Website. Dec 1, 2004. http://www.jcp.org/jsr/detail/168.jsp
[9] J. Novotny, M. Russell and O. Wehrens, *GridSphere: An Advanced Portal Framework*, EUROMICRO, 2004.
[10] M. Russell, J. Russell and O. Wehrens, *The Grid Portlets Web Application: A Grid Portal Framework*, PPAM, 2005.
[11] M. Bogdanski, M. Kosiedowski, C. Mazurek and M. Stroinski, *Facilitating the Process of Enabling Applications within Grid Portals*. Lecture Notes in Computer Science 3251: Grid and Cooperative Computing GCC 2004. Springer-Verlag. Berlin Heidelberg New York (2004) 175-182
[12] M. Bogdanski, M. Kosiedowski, C. Mazurek and M. Wolniewicz, *GRID SERVICE PROVIDER: How to improve flexibility of Grid user interfaces*, Lecture Notes in Computer Science 2657: Proceedings of the International Conference on Computing Science ICCS 2003, Springer-Verlag. Berlin Heidelberg New York 255-263 (2003).
[13] P. Grzybowski, M. Kosiedowski and C. Mazurek, *Web Services Communication within the PROGRESS Grid-Portal Environment*, Proceedings of the International Conference on Web Services ICWS 2003. Las Vegas USA (2003) 340-345
[14] M. Kosiedowski, M. Malecki, C. Mazurek, P. Spychala, and M. Wolski, *Integration of the Biological Databases into Grid-Portal Environments*, Workshop on Database Issues in Bioological Databases DBiBD. Edinburgh UK (2005), accessed from http://progress.psnc.pl/

JULIUSZ PUKACKI received his M.Sc. degree in computer science from Poznań University of Technology (Parallel and Distributed Computating). Since 1998 he is working in Poznań Supercomputing and Networking Center. At first he was employed in Network Services Department as a programmer of inetrnet and intranet services. Next he changed department to Application Department and started to work in the Grid computing area. Currently in Jarek Nabrzyski team he works on solutions for resource management in the Grid environment. Pukacki has been involved in a number of Grid projects. One of the most important was GridLab Project where he was a leader of work package responsible for resource management.He is still leading a development of GRMS (Grid Resource Management System) – metascheduler for Grid environments. Other projects are ACGT, Inteligrid, HPC-Europa, and national ones: Progress (project done in cooperation with Sun Microsystems), SGI Grid, and Clusterix. In all those project he is working on delivering resource management solutions for Grids.



MICHAŁ KOSIEDOWSKI, has been with the Poznań Supercomputing and Networking Center since 1997. He received his master's degree from the Poznań University of Technology in 1998. He has been involved in research and developments concerning web and portal solutions. Some of the major projects he participated in include the Multimedia City Guide and Polish Educational Portal. Since 2002 Michal has been involved in the design and development of solutions for construction of access environments to Grid resources and services. He is author and co-author of about 10 publications concerning Grid access and Grid portals in professional journals and major conferences.

RAFAŁ MIKOŁAJCZAK, M.Sc. [e-mail: Rafal.Mikolajczak@man.poznan.pl]. M.Sc. degree in Computer Science from the Poznań University of Technology in 1998. Currently he is employed as a HPC specialist position at the Supercomputing Department in Poznań Supercomputing and Networking Center. His research interests concern checkpoint low level services and Grid service and distributed data management. Hi is also responsible for the storage system in PSNC. He took part in the preparations of the distributed storage management project proposals (National Data Store, successfully accepted by the Polish Government). During the last three years his R&D activity concerns the checkpointing and migration issues in HPC/HTC computing. He is coauthor of "Resources virtualization in fault-tolerance and migration issues" paper that will be published in ICCS204 proceedings. In years 2001-2003 he gains the experience in checkpointing area issues by participating in implementation user- and kernel-level checkpointing mechanism for the Solaris OS on Sparc CPU. This work was done within the PROGRESS project (http://progress.psnc.pl). Currently he is working on kernel-level checkpointing mechanism for IA64 with Linux OS. This effort is being done within the SGIGrid project (http://www.wcss.wroc.pl/pb/sgigrid/en/index.php).

MARCIN ADAMSKI obtained an M.Sc. in Computer Science in 1996 with the Intelligent Decision Support Systems Department at the Poznań University of Technology (PUT). After obtaining his M.Sc. he began working for PUT, developing GUI clients to network analyzers for Siemens A.G. and Tektronix, Inc. He turned to Grid computing in 2002 when he joined the GridLab Project, helping to design and develop the Gridge Authorization Service (GAS). Now, as project lead for GAS, he is actively developing and overseeing its use in the InteliGrid Project. His current research activities are focused primarily on authentication and authorization processes in Grids and in Virtual Organizations.

PIOTR GRABOWSKI obtained an M.Sc. in Computer Science in 1998 at Poznań University of Technology – Distributed Computer Systems department. After his M.Sc., he joined the programmers group at PUT and worked on mobile network protocol analyzers software for Siemens A.G. and Tektronix, Inc. In 2002 he joined GridLab project "Access for Mobile Users" workpackage. The main thrust of his current research work are Mobile Devices, Web Services and Web technology. Piotr is working for Gridge on enabling access from Mobile Devices to the Grid.

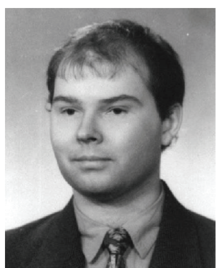
MICHAL JANKOWSKI, received his M.Sc. in computer science in 1998 from Poznań University of Technology (Parallel and Distributed Computating). 1998-2003 he worked in Poznań University of Technology on mobile network protocol analyzers software for Tektronix, Inc. Since 2003 he has been working in Poznań Supercomputing And Networking Center, Supercomputing Department and he has participated in a number of Grid projects: SGIgrid, Clusterix, Coregrid, BalticGrid. His special research interests are user management and accounting in Grids. He is an author or co-author of several papers in professional journals and conference proceedings.

MIROSLAW KUPCZYK (MK) received M.Sc. degree in Computer Science from the Poznań University of Technology (1999), Distributed Computer Systems specialization. Currently he is employed as a HPC Specialist at the Supercomputing Department in Poznań Supercomputing and Networking Center (PSNC). His research interests concern graph algorithms, grid user working environment, resource management in the grid technology and administration, configuration of HPC systems. Since 1998 MK has been responsible for putting into practice load sharing facilities (Platform Comp. LSF) on SGI and Cray machines. He has been involved in several grid projects (CrossGrid, EGEE, BalticGrid, SGIgrid with co-operation with Silicon Graphics, Polish National project PROGRESS, *etc*). He is an author and co-author of several reports and papers in scientific journals and conference proceedings.

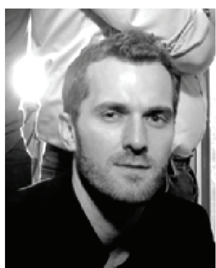
CEZARY MAZUREK, Ph.D., Head of the Network Services Department at Poznań Supercomputing and Networking Center. He received his PhD degree in Computer Science from Poznań University of Technology in 2004. His research interests concern a wide variety of advanced network services including portal solutions, digital multimedia libraries, streaming technologies, distance learning and access to grid services. He has been the manager of numerous projects in those fields coordinated by PSNC. Some of the major ones include the Multimedia City Guide, Polish Educational Portal, Digital Library Framework: dLibra, Computer recruitment to schools 2003-2005. In 2001-2004 he was leading the PROGRESS project titled "Creating an access environment for GRID computational services performed by cluster of SUNs", co-funded by the State Committee for Scientific Research and SUN Microsystems. Since 2003 he is the leader of Interactive TV project co-funded by the State Committee for Scientific Research and Polish National Public Television. He is an author or co-author of over 50 papers in professional journals and conference proceedings.

DR NORBERT MEYER is currently the Head of the Supercomputing Department in Poznań Supercomputing and Networking Center. His research interests concern resource management in the GRID environment, GRID accounting, data and storage management, technology of development graphical user interfaces and network security, mainly in the aspects of connecting independent, geographically distant Grid domains. He conceived the idea of connecting Polish supercomputing centres, the vision of dedicated application servers and distributed storage infrastructure. Norbert Meyer is co-author of the CERT Pionier organization (http://cert.pol34.pl). Leader of international and national project activities and member of Steering Groups, *e.g.* EU CoreGRID (www.coregrid.net), BalticGrid (www.balticgrid.org), CrossGrid (www.eu-crossgrid.org). He participated in several national projects concerning the HPC technology, *e.g.* being co-author of the projects entitled Creating an access environment for computational services performed by cluster of SUNs, VLAB – High Performance Computing and Visualisation for Virtual Laboratory Applications, Clusterix – National CLUSTER of LInuX Systems project. An example of a production Grid is a project proposal done in co-operation with IBM, which concerns the deployment of utility computing in Poland. He is also author and co-author of several reports and papers (60+) in conference proceedings, member of conference Programme Committees. He is also a member of eIRG (eInfrastructure Reflection Group) which is a group of experts related to European research infrastructure.

DR JAREK NABRZYSKI received his M.Sc. and Ph.D. degrees in computer science from Poznań University of Technology in POLAND. Currently he is a researcher at the Poznań Supercomputing and Networking Center (PSNC), where he heads the Applications Department. His research interests over the last 10 years have focused on knowledge-based multiobjective project scheduling, and resource management for parallel and distributed computing. For the last couple of years he has been working on tools and middleware technologies for computational grids. Jarek Nabrzyski is a co-founder of the European Grid Forum and the Global Grid Forum. In 2001-2002 he was a member of the Global Grid Forum Steering Group where he was the Area Director of the Applications, Programming Models and Environments Area. In 2002-2005 he managed the European GridLab project, in which he was one of the Principal Investigators, responsible for such areas as Resource Management, Security and Mobile User Support. He is also involved in a number of 6FP projects, including *e.g.* ACGT, InteliGrid, GridCoord, BREIN, QosCosGrid, Challengers, BeInGrid, OMII-Europe. Jarek Nabrzyski is a member of several advisory boards, including projects such as Akogrimo, CoreGrid and UCoMs (USA). He is also a member of the KISTI Supercomputing Center Advisory Board (Korea).



TOMASZ PIONTEK received his M.Sc. in computer science in 1998 from Poznań University of Technology (Parallel and Distributed Computating). 1998-2002 he worked at Poznań University of Technology and participated in Tektronix Project. Since 2002 he has been working in Poznań Supercomputing And Networking Center, Application Department as Research Programmer in various Grid projects: GridLab, HPC-Europa, ACGT. His research interests include distributed computing and resource management. Tomasz belongs to GRMS team and works on solutions for resource management in the Grid environment.



MICHAEL RUSSELL began working under the direction of Ian Foster at the University of Chicago in May 2000 where he lead development of the Astrophysics Simulation Collaboratory Portal and helped to disseminate information about the Globus Project. In May 2002 he moved to Berlin, Germany to manage the Grid Portals Work Package of the GridLab Project at the Max Planck Institute for Gravitational Physics (Albert Einstein Institute). While working on the GridLab Project, he and his colleagues created the GridSphere Portal Framework. Now, as group leader of the GridSphere development team at the Poznań Supercomputing and Networking Center (PSNC), he is helping to deliver Grid portal solutions for the Open Middleware Infrastructure Institute for Europe, HPC Europa and several other projects in Europe.

Maciej Stroiński received the Ph. D. degree in Computer Science from the Technical University of Gdańsk in 1987. Currently he is Technical Director of the Poznań Supercomputing and Networking Center. He is also lecturer in the Institute of Computing Science of the Poznań University of Technology. His research interests concern computer network protocols and management. He is author or co-author of over 100 papers in major professional journals and conference proceedings.

Marcin Wolski started working at PSNC in 2001 as a database and network programmer. In January 2002, he joined the Data Management System (DMS) team, where he worked as a system analyst and developer. In 2003, he began to lead the DMS and modified its development direction towards grid environments and large-scale solutions. He was also responsible for putting into practice the Data Management System software in the scope of SGI project. In 2005, he was in charge of developing GIS applications and services for the PIONIER network. These works were presented during Terena Networking Conference 2005 as a part of 40 Gbps pilot network. Before turning to grid solutions, Marcin Wolski had worked on traditional GUI applications, database middleware and Web-based applications. Additionally, he possesses a thorough experience in database technologies, approved by participating in Oracle trainings and official conferences. His personal interests concern system oriented architectures, autonomic computing and enterprise application frameworks.