UNICORE Summit 2010

Proceedings, 18 – 19 May 2010 I Jülich, Germany

Achim Streit, Mathilde Romberg, Daniel Mallmann (Editors)

JÜLICH
FORSCHUNGSZENTRUM

Mitglied der Helmholtz-Gemeinschaft

Schriften des Forschungszentrums Jülich
IAS Series Volume 5

Forschungszentrum Jülich GmbH
Institute for Advanced Simulation (IAS)
Jülich Supercomputing Centre (JSC)

UNICORE Summit 2010

Proceedings, 18 – 19 May 2010 I Jülich, Germany

edited by Achim Streit, Mathilde Romberg, Daniel Mallmann

Schriften des Forschungszentrums Jülich
IAS Series Volume 5

ISSN 1868-8489 ISBN 978-3-89336-661-3

Bibliographic information published by the Deutsche Nationalbibliothek.
The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data are available in the Internet at http://dnb.d-nb.de.

Publisher and Distributor:	Forschungszentrum Jülich GmbH Zentralbibliothek 52425 Jülich Phone +49 (0) 24 61 61-53 68 · Fax +49 (0) 24 61 61-61 03 e-mail: zb-publikation@fz-juelich.de Internet: http://www.fz-juelich.de/zb
Cover Design:	Grafische Medien, Forschungszentrum Jülich GmbH
Printer:	Grafische Medien, Forschungszentrum Jülich GmbH
Copyright:	Forschungszentrum Jülich 2010

Schriften des Forschungszentrums Jülich
IAS Series Volume 5

ISSN 1868-8489
ISBN 978-3-89336-661-3

The complete volume ist freely available on the Internet on the Jülicher Open Access Server (JUWEL) at http://www.fz-juelich.de/zb/juwel

Persistent Identifier: urn:nbn:de:0001-2010082304
Resolving URL: http://www.persistent-identifier.de/?link=610

Preface

The UNICORE Grid technology provides a seamless, secure, and intuitive access to distributed Grid resources. UNICORE is a full-grown and well-tested Grid middleware system, which today is used in daily production worldwide. Beyond this production usage, the UNICORE technology serves as a solid basis in many European and International projects. In order to foster these ongoing developments, UNICORE is available as open source under BSD licence at `http://www.unicore.eu`.

The UNICORE Summit is a unique opportunity for Grid users, developers, administrators, researchers, and service providers to meet. The first UNICORE Summit was held in conjunction with *Grids@work - 2nd Grid Plugtests* from 11 – 12 October 2005 in Sophia Antipolis, France. In 2006 the style of the UNICORE Summit was changed by establishing a Program Committee and publishing a Call for Papers. In the following years (2006 – 2008) the UNICORE Summit was held in conjunction with the *Euro-Par* conference. The proceedings of the events are available in Springer's LNCS series: 2006 – LNCS 4375, 2007 – LNCS 4854, 2008 – LNCS 5415, and 2009 – LNCS 6043.

In 2010 the format of the UNICORE Summit was changed to re-focus on its original aim to provide a unique opportunity for sharing experiences, presenting past and future developments and get new ideas for prosperous future work and collaborations. Therefore a Call for Contribution was published that allowed presentations as well as demonstrations. Furthermore the UNICORE Summit 2010 was organised as a stand-alone event and took place 18 – 19 May at the Jülich Supercomputing Centre in Jülich, Germany.

Prof. Geerd-Rüdiger Hoffmann from DWD opened the event with an historical overview about the beginning of UNICORE more than 12 years ago. The invited talk about EMI – The European Middleware Initiative – was given by Dr. Alberto Di Meglio (CERN), project director of EMI. The technical part consisted of 17 contributions split-up in 10 presentations and 7 demonstrations. The proceedings at hand include a selection of 14 papers that show the spectrum of where and how UNICORE is used and further extended.

The large amount of contributions received as well as over 40 international participants demonstrated the success of the new format.

Finally, we would like to thank all authors and presented for their contributions, camera-ready versions, and presentations at the UNICORE Summit 2010 in Jülich.

More information about the UNICORE Summit series can be found at `http://www.unicore.eu/summit`. We are looking forward to the next UNICORE Summit!

August 2010

Achim Streit
Mathilde Romberg
Daniel Mallmann

Contents

Preface
A. Streit, M. Romberg, D. Mallmann **i**

A UNICORE-based Multi Site and Multi User Grid Environment for Demonstration, Education, and Testing Purposes
T. Kálmán, Th. Rings **1**

Building UNICORE Based Desktop Grid
J. Jurkiewicz, P. Bała **11**

DEISA Development Environment
G. Fiameni, M. S. Memon, S. H. Leong, X. Pegenaute, J. Jimenez, C. Gheller, R. Brunino, D. Mallmann **19**

Secure High-Throughput Computing Using UNICORE XML Spaces
R. Grunzke, B. Schuller **27**

Synergizing ETICS and UNICORE Software
A. Giesler, A. Streit, E. Ronchieri, M. Dibenedetto **37**

HiLA 2.0 – Evolutionary Improvements
B. Hagemeier, R. Menday **45**

MMF: A Flexible Framework for Metadata Management in UNICORE
W. Noor, B. Schuller **51**

Extended Execution Support for Scientific Applications in Grid Environments
B. Demuth, S. Holl, B. Schuller **61**

Running License Protected Applications with SmartLM and UNICORE
H. Rasheed, A. Rumpl, W. Ziegler, B. Hagemeier, D. Mallmann, H. Eickenbusch **69**

Resource Usage Accounting for UNICORE
P. Bała, K. Benedyczak, M. Lewandowski **75**

The Key Role of the UNICORE Technology in European Distributed Computing Infrastructures Supporting e-Science Applications in the Decades to Come
M. Riedel, M. S. Memon, A. S. Memon, S. Holl, D. Mallmann, N. Lamla, A. Streit, Th. Lippert **83**

Experiences with UNICORE 6 in Production
M. Rambadt, S. Bergmann, R. Breu, N. Lamla, M. Romberg **95**

UNICORE Runtime Administration
M. J. Daivandy, B. Schuller, B. Demuth **105**

Monitoring of the UNICORE middleware
P. Bała, K. Benedyczak, M. Strzelecki **117**

A UNICORE-based Multi Site and Multi User Grid Environment for Demonstration, Education, and Testing Purposes

Tibor Kálmán[1] **and Thomas Rings**[2]

[1] Gesellschaft für Wissenschaftliche Datenverarbeitung mbH,
Am Fassberg 11, 37077 Göttingen, Germany
E-mail: tibor.kalman@gwdg.de

[2] Institute of Computer Science, University of Göttingen,
Goldschmidtstrasse 7, 37077 Göttingen, Germany
E-mail: rings@cs.uni-goettingen.de

This article gives an overview of the novel Instant-Grid UNICORE edition. The Instant-Grid UNICORE edition allows to automatically set up a self-configured and independent multi site and multi user grid environment using the UNICORE 6 middleware. The idea of such an environment is to enable the installation of a grid system on computers located in a local network without any previous knowledge of grid technologies. In this article, we describe the technical concepts including the automatic configuration, ready-to-use features, and applications of the Instant-Grid UNICORE edition. In addition, we consider new features and present an use case where Instant-Grid has been applied in teaching. From our experience, Instant-Grid UNICORE edition is also suitable for demonstrating, developing, and testing purposes of and in grid computing environments.

1 Introduction

A grid is a networked system "that coordinates resources that are not subject to centralized control using standard, open, general-purpose protocols and interfaces to deliver nontrivial qualities of service"[3]. It provides a service oriented environment for sharing computational and storage resources distributed all over the world and belonging to diverse organizations. Grid computing offers a way to solve large scale computational and storage challenges including applications for protein folding, financial modeling, or earthquake simulation.

However, the establishment and setup of a grid environment is a non-trivial task that requires deep knowledge of grid computing technology. Therefore, providing a solution such as the Instant-Grid[7] UNICORE edition that automatically establishes a trustful multi user grid environment increases the acceptance of grid technologies. In this article, we present the novel Instant-Grid UNICORE edition that automatically sets up a self-configured and independent standalone grid based on the *Uniform Interface to Computing Resources* (UNICORE) 6[12] on computers located in a local network. The idea of this environment is to allow setting up a grid system without any pre-existing knowledge of grid technologies.

A grid system established in such an ad-hoc way can then be used for demonstration, education, development, and testing purposes. For grid developers, it provides ready-to-use tools and a production-independent grid test environment. As a closed grid system, it supports tests of grid components and applications. In addition, Instant-Grid includes

pre-installed grid applications and fully configured services. This means that these applications automatically make use of all the available grid resources. The grid applications are configured inside the grid security environment and can be used via a web browser. Therefore, Instant-Grid UNICORE edition is also ideally applicable for education purposes. The students can configure, and administrate their own grid system and also develop, use, and test grid applications and their distribution.

The Instant-Grid Project was started in 2005 and was funded by the German Federal Ministry of Education and Research until 2007. Initially, Instant-Grid is based on Globus Toolkit 4[4]. After 2007, further developments were driven by the Instant-Grid community. End of 2009, the development of an Instant-Grid based on the UNICORE 6 middleware providing all characteristics described above was started. The Instant-Grid UNICORE edition applies to the specific requirements of UNICORE 6 and provides an ideal interoperability test environment.

This article is structured as follows: In Section 2, we describe the technical concepts of Instant-Grid. Afterwards in Section 3, new features of the Instant-Grid UNICORE edition are presented. In Section 4, we describe the application of Instant-Grid in a practical course at the University of Göttingen. In Section 5, we consider related work. Finally, we conclude with a summary and an outlook in Section 6.

2 Technical Concept

The Instant-Grid UNICORE edition automatically sets up a self-configured grid system by booting a dedicated frontend from *Compact Disc* (CD) or *Universal Serial Bus* (USB). The grid system is based on UNICORE 6 and located on computers in a local network. The booted frontend, i.e., the Instant-Grid server is responsible for a network-based boot of other participating computational nodes that act as UNICORE Sites. The Instant-Grid frontend provides mechanisms to discover all changes in the local grid system at run-time, e.g., the removal or addition of resources and also updates several resource databases utilized in Instant-Grid. These capabilities allow using preinstalled applications and features in the Instant-Grid environment.

This section describes how Instant-Grid UNICORE edition combines the Live-CD concept with the *Preboot Execution Environment* (PXE) network boot mechanism. We describe the fully automated service- and network-setup during the boot process. Afterwards, we consider enhanced functionalities such as the automated discovery, service configuration, and user management that work at runtime. Finally, we present ready-to-use grid features and applications.

2.1 Automated Configuration at the Startup Process

Technically, services to turn a computer lab into a grid system are based on the well established PXE mechanism, which is also used by the original Knoppix on which Instant-Grid is based as well as in Live-CD projects aiming at cluster setups. After booting the Instant-Grid image from a physical volume of a desktop computer, fully automatically configured servers for *Dynamic Host Configuration Protocol* (DHCP), *Trivial File Transfer Protocol* (TFTP), and *Network File System* (NFS) run. The DHCP server is already configured to answer PXE boot requests from client nodes. Then, the kernel and needed boot files are

transferred via TFTP. After booting the kernel, the nodes mount their root filesystem via NFS from the frontend. This startup process and all servers are configured automatically and do not require any human intervention. The configuration steps that are done at the grid level are described in section 3.1.

In addition, an automatic search for a suitable network interface on the frontend node as well as on the clients is performed. The frontend node can directly access the Internet through an optional external interface. On the client nodes all interfaces, except the one for the internal network, are disabled. In case an external interface is set up on the frontend node, the nodes can access the Internet via *Network Address Translation* (NAT). To the best of our knowledge, all other implementations using PXE for building a cluster need further manual configuration steps before starting all required services.

2.2 Automated Configuration at Runtime

An important idea about grid computing environments is that the resources are assumed to be dynamic. This means, client nodes can be started or stopped at any time the grid system exists. In case of Instant-Grid, the grid environment exists as long as the frontend node runs. To accommodate this idea, a mechanism has been implemented to discover new and lost nodes. This information is then distributed to all participating nodes. In Instant-Grid we developed such a mechanism for the frontend by probing the network for available nodes, updating the configuration, and provide it to all nodes via a specific exported NFS directory. On the client side, this directory is mounted and periodically checked in a preconfigured interval. In case of changes, the local configuration is updated.

The dynamic character of a grid system - as opposed to the static character of a cluster — necessitates the availability of an information service, which provides current data on status, resources, and services of the grid. In Instant-Grid, the hardware resources are monitored using the Ganglia Monitoring System[5], while the web service based *Common Information Service* (CIS)[2] of the UNICORE 6 handles information about available services. Both, Ganglia and CIS are started and configured without any user involvement at boot time. Additionally, Instant-Grid provides test routines to ensure the correct behavior of the registered resources and services. The availability and load information obtained by Ganglia can also be seen in a web frontend.

2.3 Ready-to-Use Features and Applications

The default installation of Instant-Grid results in a set of ready-to-use grid functions and applications. They were chosen to be understandable for non-experts. The examples demonstrate the uses and benefits of grid environments and support the development of grid applications. The grid applications are preconfigured for demonstrations and work inside the local Instant-Grid environment. Neither Internet connection nor global grid connectivity is required.

The *Persistence of Vision Raytracer* (POV-Ray) application creates photo-realistic 3D images using a rendering technique called raytracing. This is a CPU intensive task, but can easily be parallelized. POV-Ray is one of the applications that was chosen to demonstrate the dynamic resource management of a grid system. Each participating Instant-Grid client renders a part of the original 3D image or one frame of an animation, which can be done simultaneously. Here, Instant-Grid is used as a renderfarm for rendering complex scenes.

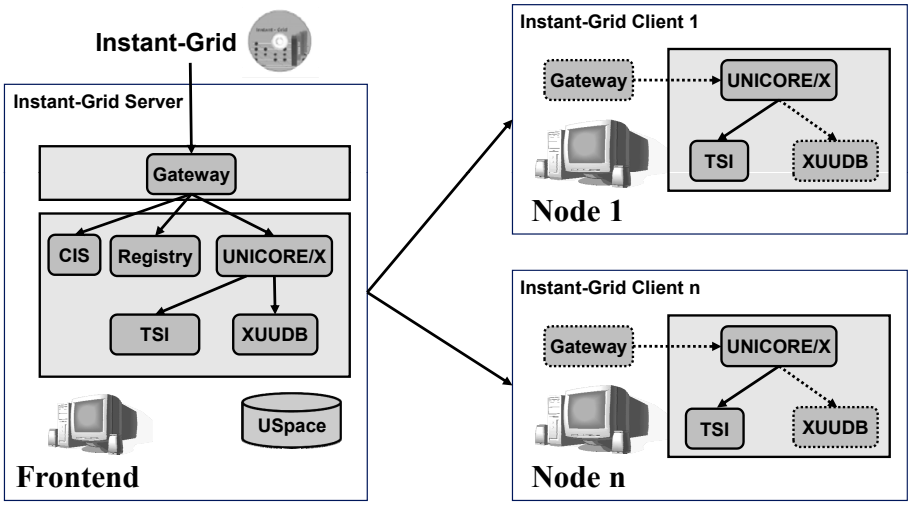

Figure 1. Dynamically configured UNICORE 6 architecture of Instant-Grid

Certain applications and tools have been chosen and integrated in the Instant-Grid environment to demonstrate the possibilities of collaborative works in grid environments. These include a realtime collaborative editor and a chat client with a built-in whiteboard. In addition, the GridSearch framework for building an index of text corpora is integrated. Whenever a new resource, like a CD or an USB-stick is introduced into any of the clients of the Instant-Grid environment, the texts on the media can be indexed. The index data is then accessible for other user that collaborate in the grid.

3 New Features of the UNICORE 6 Edition

This section describes several new features of the Instant-Grid UNICORE 6 edition. These include the dynamic configuration of the UNICORE 6 components on every node, the setup of the UNICORE security environment, and customization possibilities such as the nightly-build environment of the Instant-Grid community and the persistent setup on demand.

3.1 Dynamic Configuration of UNICORE 6

In addition to the configuration of basic services during the startup process as described in Section 2.1, certain dynamic configurations of UNICORE 6 components need to be applied in the startup on every nodes. This allows an easy deployment of the Instant-Grid UNICORE edition environment without pre-existing knowledge of UNICORE or grid technologies.

The dynamically configured UNICORE 6 architecture of Instant-Grid is depicted in Figure 1. The frontend, i.e., the Instant-Grid server is responsible for the management of the nodes including hostname allocation, signing and issuing of host certificates, and for the correct setup of the UNICORE components.

In an Instant-Grid environment, a UNICORE Gateway is configured to provide access to services and to perform authentication decisions. Because of performance issues, a UNICORE Gateway is not started on the clients. The UNICORE services running on the client nodes are served by the Gateway that runs on the frontend. The service address of the Gateway is distributed to all clients in order to allow services to contact the gateway.

A global, shared Registry is started on the Instant-Grid frontend. The service address of the Registry is communicated to the other services in order to register all services. The UNICORE 6 services that run on the Instant-Grid nodes are automatically configured and registered in the UNICORE registry without any user intervention.

The central component for job and data management inside UNICORE 6 is UNICORE/X which is a container for the UNICORE 6 atomic services and also includes a local Registry service. UNICORE/X is started on all Instant-Grid nodes. However, the auto registration with an external registry is enabled for the services on the clients. They are able to contact the shared Registry on the frontend in order to publish and query information. Other UNICORE/X parameters are also configured automatically during the startup. For example, the hostname is used as the name of the UNICORE/X site and by the AdminDomain and ComputingService entities of the site description. The site description is used by the *Common Information Provider* (CIP) which collects resource information from the UNICORE/X and constructs a *Grid Laboratory Uniform Environment* (GLUE) v2.0 compliant XML document for the information service.

The grid information service CIS is also dynamically configured and started on the Instant-Grid frontend. It fetches all connected Instant-Grid sites from the global Registry and periodically gathers static and dynamic resource information from all nodes. It aggregates and publishes the resource information in GLUE v2.0 format and provides XPath and XQuery standard query mechanisms. The CIS in Instant-Grid stores the resource information in a native XML database. The web interface of CIS is also set up to enable searching and browsing entities published via GLUE v2.0 documents.

In addition to the Gateway which is responsible for the authentication, a *UNICORE User Database* (XUUDB) service which performs authorization in the Instant-Grid environment is executed. The grid certificates of the local Instant-Grid users are registered in the XUUDB database and mapped to a local Linux account automatically. Because of performance issues, the XUUDB service is not started on the clients. The service address of the XUUDB running on the frontend are dynamically configured on all clients during the boot process and the client nodes are served by that XUUDB service.

The job directory of UNICORE, the USpace, exists in ramdisk only. It can be configured on a local harddisk on the frontend. The USpace is then shared via NFS and accessible to the clients.

The *Target System Interface* (TSI), which is an interface to batch systems and allows job submission into local resource management systems, is under development. However, the embedded TSI starts jobs as a simple unix fork job.

The services with dotted frame in Figure 1) do not run because of performance issues, but can be enabled for demonstration purposes. For example, the XUUDB service should be enabled on each node, to establish a multi-user and a multi-site Instant-Grid environment.

Two UNICORE clients are installed in Instant-Grid and can be used to access the services after the grid environment is started. The *UNICORE Command Line Client* (UCC)

is a command-line tool which can be utilized in a shell or scripting environment. Additionally, the ucc extensions for CIS commands are preconfigured and the CIS libraries are deployed in the Instant-Grid UNICORE edition.

The *UNICORE Rich Client* (URC) is an Eclipse-based graphical client, which provides a graphical view of the local Instant-Grid environment. All registered UNICORE resources in a running Instant-Grid environment can be browsed. The resources can be used to execute grid application utilizing these resources. Resource requirements, e.g., required main memory or number of processors can also be specified for jobs.

3.2 Security Environment for UNICORE

The UNICORE security environment is based on the X.509 public key infrastructure. Therefore, an Instant-Grid certificate authority is initialized on the frontend as a part of the boot process. It is based on the Open Source OpenSSL toolkit and issues user and server certificates. Certificates identifying trusted parties are stored in a truststore. UNICORE 6 requires truststores in *Java Key Store* (JKS) format. The Instant-Grid certificate authority certificate generated during the startup process is included in every truststore. Private keys belonging to users or services are stored in a keystore, which can be *Public-Key Cryptography Standards #12* (PKCS12) or JKS format. The necessary key and trust stores are automatically created if a new user account is required by a new user, or a new client requires a host or service certificate.

3.3 Customization of Instant-Grid

Configurations applied in a running Linux Live CD system are not persistent. Usually the system starts with default settings and But for specific conditions such as in case for teaching, a customized Instant-Grid is required. Therefore, Instant-Grid supports two types of persistent configurations that are needed, e.g., for user or software management. These are described in the following.

3.3.1 Configuration in the Image Source

A CD image can be customized by building a modified version of it. Changes are directly applied in the source from which the CD image is built. This process, also call remastering takes place every night because it is CPU and time intensive. A new version with the changes applied by the community in the day before is then produced automatically. It is also released to the developers as a test version every night in order that the modifications can be verified. The advantage of Knoppix is that it uses the specific Cloop compressed filesystem. In contrast to usual Live CD systems, Cloop makes it possible to integrate about 2 *Gigabyte* (GB) software on a 700 *Megabyte* (MB) CD image.

In addition, binary Debian software packages are built to automate the management of additionally required software components. To create a debian package, the data part is downloaded (cached) directly from the UNICORE software page, and the control information is created by the Instant-Grid developers. Several legacy Debian packages have been built for several UNICORE software components such as the UNICORE 6 Server, CIS, UCC, and RichClient. For a consistent setup within Instant-Grid, an additional configuration package is built.

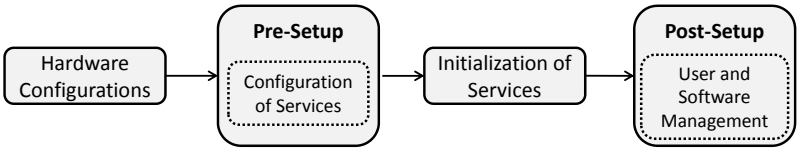

Figure 2. New setup phases during boot of the Instant-Grid UNICORE edition

These UNICORE software packages are then copied to the Instant-Grid software repository. The package management system of the operating system evaluates the meta-information to allow package searches, to perform an automatic installation, or to check that all dependencies of a package are fulfilled. The selected packages and their dependencies are installed automatically into the Live CD in a chroot environment.

The Instant-Grid image does not contain any configurations, e.g. hostname or certificates, because these are dynamically configured during the boot process.

3.3.2 Fast Adaptable Persistent Setup

Creating a Live CD image is a time, storage, and CPU intensive process. Therefore, we suggest to store specific and customized configuration data and scripts in a directory on the filesystem. This directory is mounted during the boot process and parsed for scripts which are then executed to apply certain configurations. This method enables a fast, local, and persistent configuration. However, it can only work on the ramdisk and is, therefore, quite expensive. Also, the data is not written permanently. As depicted in Figure 2, we determined two configuration phases: the pre-setup phase and the post-setup phase which are integrated in addition to the fully automated setup as presented in Section 2.1.

After the hardware components are fully configured and all necessary disks are mounted, the pre-setup phase is invoked. This phase needs to be executed before services are started by the init-process. The pre-setup is suitable to change the default service configuration of Instant-Grid. For example, X.509 certificates used by the different services are generated during every boot process. The pre-setup stage can be used to overwrite the default behavior for its generation and use specific service certificates issued by other certificate authorities. Another example is to change specific service ports in case they are not suitable for the local network environment or its administrative restrictions.

The post-setup phase is invoked after all components are set up and all services are started. It can be used amongst others to add further grid users or install software packages that are not included in the Instant-Grid image.

User applications or data can also be integrated by using a specific local and persistent filesystem. The filesystem is available on each computing node via the NFS, and can therefore be used by grid applications. Within such a local trustful environment, multiple users are able to start, build, and use an experimental grid system at any time without the restrictions that are associated with a remote grid environment.

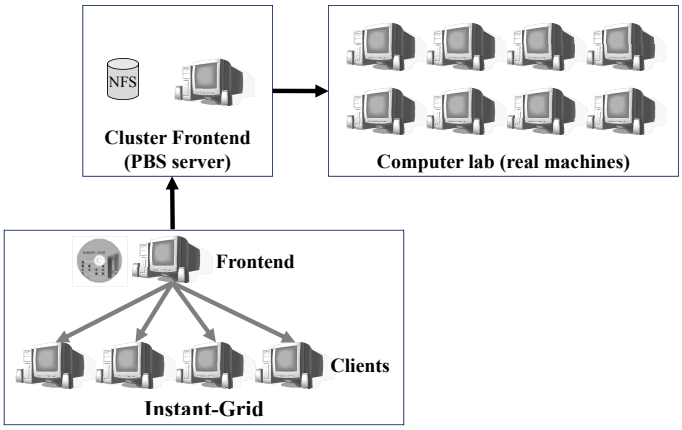

Figure 3. Instant-Grid environment deployed in the practical course

4 Use Case: Practical Course

Instant-Grid UNICORE edition offers a multi site and multi user grid environment for education purposes. The recent version of Instant-Grid is used by a practical course on grid computing given at the University of Göttingen[6,9]. Instant-Grid has been proven as an ideal tool for the technical support of the course. Instant-Grid is an adjustable and scalable instrument for ad-hoc grid applications. For short-term development, extensions such as specific portals and even complete development environments can be included. This environment allowed the students to study and use a grid on their own computers without the restrictions that are associated with a remote production grid environment.

The Instant-Grid environment deployed in the practical course is depicted in Figure 3. The Instant-Grid server is able to span several Instant-Grid clients which themselves can be used as an UNICORE computational resource. The Instant-Grid server is also configured to use a local resource manager, i.e., a *Portable Batch System* (PBS) that utilizes the computers of our lab as a computing cluster. With this configuration, the students were able to learn about grid computing with a bottom-up approach. This means that the first part of the practical course is about cluster computing and the second part on top of cluster computing about grid computing.

5 Related Work

The UNICORE community provides a preconfigured client bundle that supports easy access to a UNICORE 6 demo site for new and interested users to try out and test a grid system based on UNICORE 6. This so called UNICORE 6 Testgrid[10] includes the latest version of all UNICORE 6 components. After registering, users gain access to specific resources of the UNICORE 6 demo site with a newly generated certificate. However, users are only allowed to submit jobs with limited requirements, e.g., the jobs need to have low resource consumption and short duration. In addition, the resources are in another administrative domain that is required to trust. Instant-Grid overcomes the limitations of the

UNICORE 6 Testgrid because it does not have any restrictions on job resource consumption or job duration. In addition, all the resources are in the home administrative domain and can be trusted.

The UNICORE 6 Live CD[11] provided by the UNICORE community is a Live Linux system with all the UNICORE components already installed. UNICORE 6 Live CD contains preinstalled key- and truststores which remain the same in the CD image for each user. In the development of the UNICORE Live CD, all services are statically preconfigured. The UNICORE 6 Live CD is primarily designed for a single user on a single computer. Instant-Grid allows persistent multi user configuration and automated, multi hosts deployment of the UNICORE grid middleware. Within this environment, it is also possible to obtain administrative access to configure the grid computing environment according to specific user requirements.

In contrast to the Instant-Grid UNICORE edition, the Instant-Grid Globus Toolkit edition[1] deploys the Globus Toolkit 4 middleware. The principle of the automatically configured network remains the same in both editions. However, the differences are revealed in the deployment of the security environment and the different middleware services, which rely on different architectures. The new persistent setup and customization features described in this paper are also completely missing in the Globus Toolkit edition.

6 Summary and Outlook

In this article, we presented the novel Instant-Grid UNICORE edition that allows to build a multi site and multi user grid environment. The main benefit of the Instant-Grid UNICORE edition is that it allows to establish a grid computing environment based on UNICORE 6 without any pre-existing knowledge of grid technologies or the UNICORE system. In this article, we described the main technical realization behind such a grid environment. This includes a description of the automatic configuration setup and ready-to-use features of such a grid environment. We consider new features such as the customization and dynamic configuration of the grid environment.

The Instant-Grid UNICORE edition can be used for demonstration, education, and testing purposes in grid computing systems. As a use case, we described a practical course of grid computing where Instant-Grid builds the technical base. Within such an environment, students are able to develop grid services and applications, but also to establish, configure, and administrate their own grid system without interrupting any production grid environment. In addition, it provides a closed testing environment for grid computing applications and systems[8]. It can be used and deployed for the demonstration of the grid technology without knowing the complex constrains of grid environment. It is easy deployable without impact on the local machines.

As future work, we consider to install more demo applications, e.g., for disaster management or financial modeling where the advantages of grid can be easily identified. These demo application should be configured automatically to use UNICORE resources inside the Instant-Grid. Some UNICORE 6 components, e.g. *High Level API for Grid Applications* (HiLA) 2.0 or Workflow Engine, are missing from the Instant-Grid UNICORE edition now. After experimenting with these components they will also be included.

Tools for grid developers need also to be integrated and should provide a programming paradigm for grid computing so that the available resources can be utilized by developed

programs. In addition, a grid scheduler is required that schedules grid job automatically to Instant-Grid resources.

Furthermore, we consider to integrate a mechanism for handling overload capacity by providing a mechanism to integrate external cluster resource management systems. In addition, global connection to other grid sites is planned in order to allow the use of additional external resources. Furthermore, the Instant-Grid UNICORE edition and the Instant Grid Globus Toolkit edition will be combined, so that the user can choose the grid middleware.

References

1. C. Boehme, T. Ehlers, J. Engelhardt, A. Félix, O. Haan, T. Kálmán, B. Neumair, U. Schwardmann, D. Sommerfeld, *Instant-Grid: Fully Automated Middleware-Deployment Using a Live-CD* In International Conference on Networking and Services (ICNS'06), pages 70, IEEE Computer Society, Los Alamitos, CA, USA, 2006
2. A. S. Memon, M. S. Memon, Ph. Wieder, B. Schuller, *CIS: An Information Service based on the Common Information Model*, Proceedings of 3rd IEEE International Conference on e-Science and Grid Computing, Lillehammer, Norway, IEEE Computer Society, December 2007, ISBN: 0-7695-3064-8, pp. 465-472.
3. I. Foster, *What is the Grid? A Three Point Checklist*, Grid Today **6**, 22, 2002.
4. I. Foster, *Globus Toolkit Version 4: Software for Service-Oriented Systems*, in Proceedings of the IFIP International Conference on Network and Parallel Computing (NPC05), ser. LNCS, vol. 3779. Springer, 2005.
5. Ganglia Monitoring System, `http://ganglia.sourceforge.net/` fetched on 30.06.2010.
6. J. Grabowski, A. Quadt, B. Neumair, T. Rings, T. Kálmán, P. Weber, J. Meyer, *Praktikum: Anwendung und Programmierung im Grid*, `http://www.swe.informatik.uni-goettingen.de/edu/` fetched on 22.06.2010.
7. Instant-Grid Project, `http://instant-grid.org/` fetched on 22.06.2010.
8. T. Rings, H. Neukirchen, J. Grabowski, *Testing Grid Application Workflows Using TTCN-3*, Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation (ICST 2008), April 9-11 2008, Lillehammer, Norway, IEEE Computer Society, April 2008, pp. 210-219.
9. T. Rings, A. Aschenbrenner, J. Grabowski, T. Kálmán, G. Lauer, J. Meyer, A. Quadt, U. Sax, F. Viezens, *An Interdisciplinary Practical Course on the Application of Grid Computing*, Proceedings of the 1st Annual IEEE Engineering Education Conference (EDUCON 2010), April 14-16 2010, Madrid, Spain, ISBN: 978-1-4244-6571-2, IEEE, 2010.
10. UNICORE 6 Testgrid, `http://www.unicore.eu/testgrid/` fetched on 22.06.2010.
11. UNICORE 6 Live CD, `http://www.unicore.eu/download/unicore6/` fetched on 22.06.2010.
12. Uniform Interface to Computing Resources (UNICORE) Project, `http://www.unicore.eu/` fetched on 23.06.2010.

Building UNICORE Based Desktop Grid

Jakub Jurkiewicz[1] **and Piotr Bała**[1 2]

[1] Interdisciplinary Centre for Mathematical and Computational Modelling,
University of Warsaw, Al. Żwirki i Wigury 93, 02-089 Warsaw
E-mail: {j.jurkiewicz, bala}@icm.edu.pl

[2] Faculty of Mathematics and Computer Science
Nicolaus Copernicus University, Chopina 12/18, 87-100 Toruń

In this paper we present implementation of a simple desktop grid based on the UNICORE middleware. The desktop grid allows to run different tasks at the computing nodes providing them dynamically through UNICORE communication mechanism. The desktop grid provides high security for the simulations minimizing intrusion to the desktop computers. The system implements full UNICORE 6 security and privileges model. The biggest advantage of such system is that it is based on the standard UNICORE components, eg. it does not require modifications to the UNICORE. This allows for an easy migration to a new version of underlying middleware. Parts of the systems which are not available in the UNICORE distribution has been written using standard UNICORE API, which allows for seamless integration with the basic middleware. In order to run simulations on the desktop computers we have developed https proxy service, which allows nodes hidden behind NAT and firewall act like a servers. Another important part of the system is set of services running on the desktop PC to turn UNICORE on and off. Based on the BOINC idea of using screen saver as a tool for sending messages to other software components we have developed software to activate and deactivate UNICORE services. The UNICORE desktop grid allows for easy creation of the large distributed infrastructure Here we present details of the architecture as well as efficiency tests and performance results.

1 Introduction

Desktop grids becomes more and more popular in the world of computations. The basic infrastructure is relatively cheap - there is no expenses in buying new computers since it is based on the existing ones. The unused cycles of the desktop computers can be used[1–3].

There are many very community grids middlewares available like Boinc[4] and XtremWebCH[5]. Based of them there were build and used numerous desktop grids; SZTAKI grid[6] is one of the recent examples.

At this point it is good to distinguish between community and desktop grids. A community grids is usually worldwide, works at the Internet scale, and computing nodes are usually owned by completely independent people donating voluntary resources to the project. Desktop grids are rather at the scale of institution or organization. Usually there is a centralized management of resources. Access to the unused cycles has to fulfil rigorous security directives. The most important issue is to disallow working nodes from storing downloaded data (eg. input to the simulations) on the disc which minimizes security threats. The another problem is validity of the returned results. Even in the single organization, there is some possibility to replace working node software with the modified version which can provide fake results. Up to now there is no good solution; desktop grids usually allow for basic security but it is to weak to solve the problem.

The ideal solution is to create a group of trusted computers to process data that should not be sent outside the institution - eg. financial or medical data. Processing is than per-

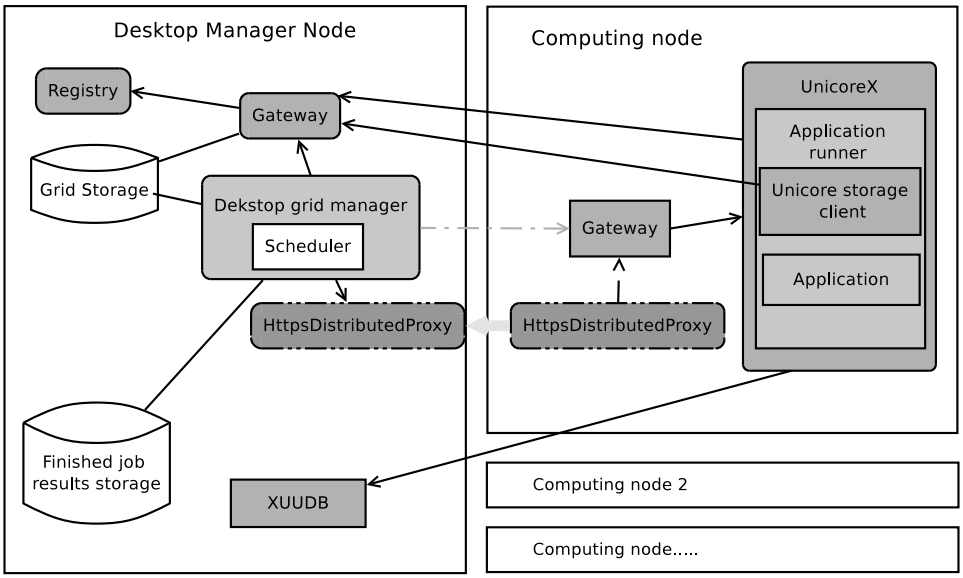

Figure 1. A schema of the UNICORE desktop grid. Intensity of grey on the schema marks types of components. From the lightest to the darkest: the lightest marks connection being tunnel for https distributed proxy. A little bit darker identifies elements created specially for desktop grid. Next intensity enhances part taken from Unicore with no change. The darkest one marks software created for https distributed proxy.

formed using computers which can be controlled within organization. Computing nodes belonging to the trusted group provide one more advantage: there is no need of checking if node returns bad results since we can trust software installed there.

One of the biggest issue of desktop and community grids is a possibility of connecting them to service grid, especially when it is placed behind firewalls and NATs. There were some projects trying to do this[7], however it was not possible to make this connection without special interface. The UNICORE 6 middleware as it adopts grid services architecture allows to solve this problem.

In this article we present UNICORE desktop grid built of standard UNICORE components. Missing parts have been developed and integrated with the main middleware. The UNICORE desktop grid has two big advantages: it allows for easy connection to the UNICORE based service grid, and provides security at the level required for running production jobs.

2 Modules of Desktop Grid

The desktop grid is built from two main components: manager server (usually one per installation) and computing node (many) - detailed schema is presented in the picture 1. The manager server is built upon standard UNICORE 6 components:

- Gateway - it is obligatory in the UNICORE setup and provides access to the other components;

- Registry - central part of manager used to store information about available resources including information about working nodes and services provided;
- Storage - used by the desktop grid manager for data storage including input files for computations and results.
- XUUDB - user database which stores authentication data for users and UNICORE components and services.

The manager server is using UNICORE Command Line Client library for node monitoring, splitting and merging jobs, submitting subtasks and collecting results. The manager server is accomplished by the https distributed proxy used for communication with nodes that are hidden behind firewalls and NAT services. The distributed proxy is described in the section 3.

The manager services are run on a single server, but there is no problem to distribute them on the different systems to increase performance and avoid network problems.

Computing node is build of following UNICORE components:

- gateway - for accessing modules of UNICORE.
- UnicoreX - used for running subtasks.

The UnicoreX, UNICORE execution service, provides special type of applications adjusted to the desktop grid needs. The application is executed as a specified Java class which downloads data from the UNICORE storage and uploads results to the UNICORE storage run by the desktop grid manager. Because of that, application run at the execution node does not need access to the local disk space. Local disk space can be used for debugging and accounting purposes such as storage of the input and error streams. This functionality is manageable and can be easily turned off.

The computing node is accomplished with the dedicated https proxy module.

The desktop grid execution model requires simulation automatic switch on and of depends on the local activity at the working node. For the Linux systems this can be easily achieved by the monitoring of the node parameters such as load and console activity. For the Windows systems this task is performed in more complicated way and consists of 3 main components. First one is Windows service which listens on the dedicated TCP/IP port for steering connection. It turns on and off second part - Desktop Grid node. Finally, start and stop commands are generated by the screen saver. The architecture of the system is presented in the picture 2, in general it is based on the architecture of BOINC community grid node[4].

3 Https distributed proxy and the communication

In the desktop grid computing nodes are usually located behind a firewall and a NAT. To overcome this problem and to be able to connect to the UnicoreX server started on the computing nodes we have designed https distributed proxy. It consists of two components: manager and node part. The manager part is visible by the clients while the node's part connects to the manager and registers addresses of the connected working nodes. The operation of the https proxy is described in the picture 3, where the communication sequence

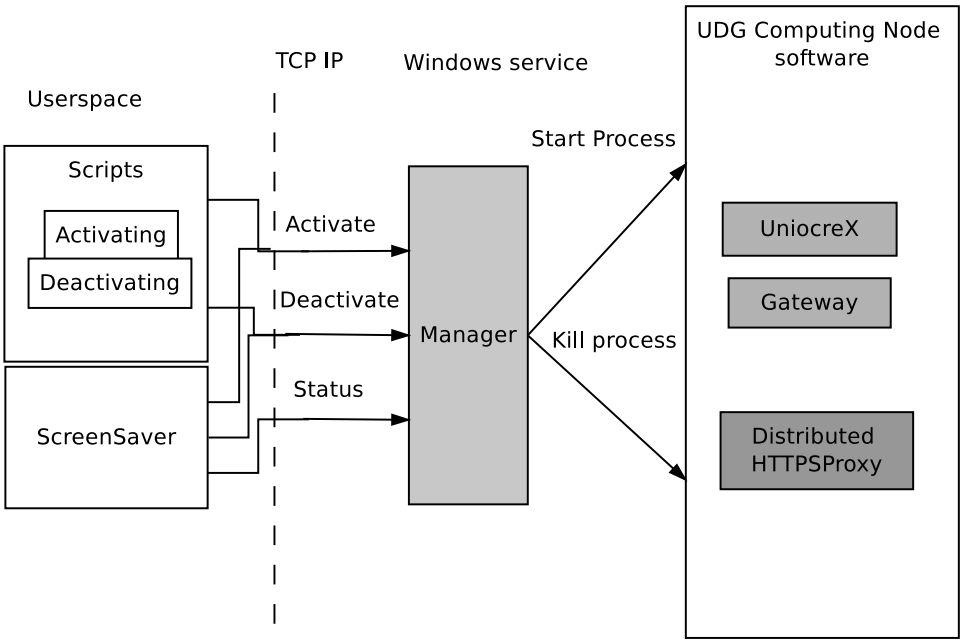

Figure 2. Schema of Windows desktop grid node.

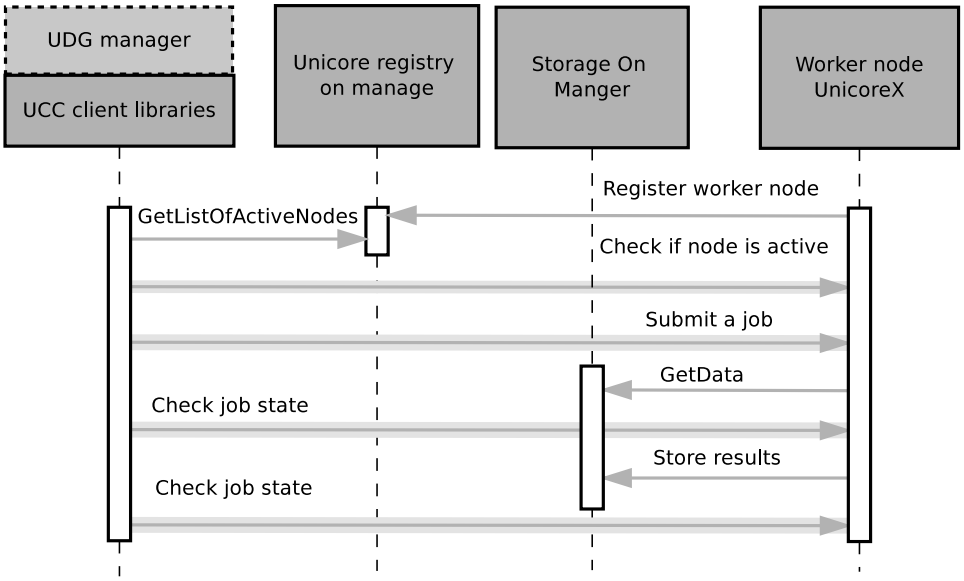

Figure 3. Communication between components of the Unicore desktop grid

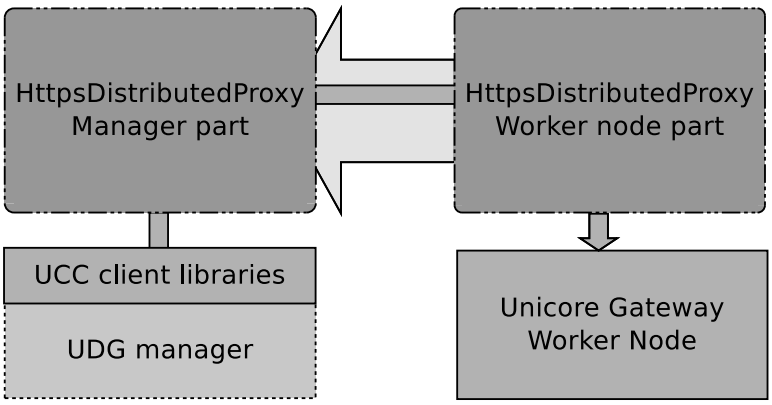

Figure 4. Https distributed proxy - connection from the desktop grid manager to the working node.

diagram is presented. The first step in this process is registration of the computing nodes at the manager services. During this UnicoreX components register in the global registry and node's part of the proxy connects to the manager.

This communication is initialized by the desktop grid manager and uses https distributed proxy tools. The communication is performed over created tunnel as presented in the picture 4

While desktop grid manager gets a job to do, it splits it into subtasks, and using tunnel sends to the computing nodes. Simultaneously it transfers data needed by the subtasks to the global storage. When task is submitted working node gets data from the global storage and when subtask is done the node sends the results back to the central storage.

The desktop grid manager monitors nodes activity by checking state of processed job. In the case of no job running, simple date job is executed.

4 Security

Since security and trust are important for the production desktop grids, we have used UNICORE security model based on the X509 certificates. When a subtask is submitted to a node, it gets its own certificate, that would allow access to the data required for computations. This certificate is also used for storing results.

Manager uses its own certificate for job submission nodes, and nodes posses their own certificates used for the communication with the other components of the UNICORE grid. Currently all nodes use the same certificate, but group of nodes could get dedicated certificates. This would allow for running jobs on nodes that are owned by the particular institution and will protect processed data. For making desktop grid work more elastic we have considered usage of the UNICORE implementation of the explicit trust delegation. Our system allows also to use normal Unicore Grid execution sites as desktop grid working nodes of desktop grid.

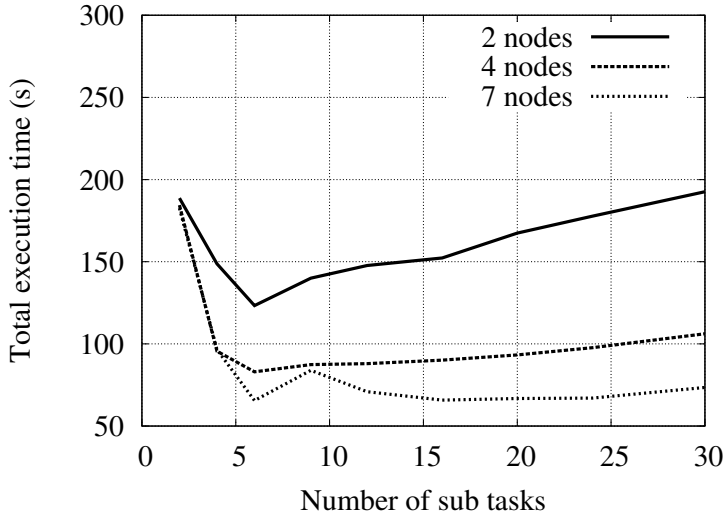

Figure 5. Efficiency test - constant number of nodes.

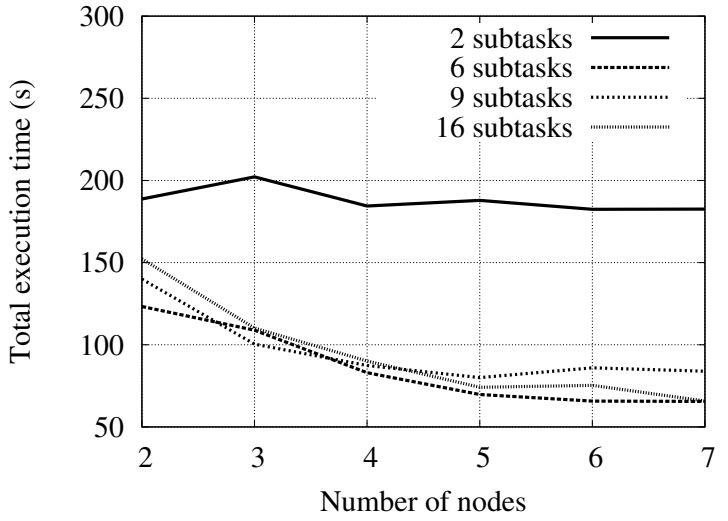

Figure 6. Efficiency test - constant number of subtasks.

5 Efficiency test

In order to check built desktop grid infrastructure we have performed efficiency test. As the workload we have used naive algorithm for distributed computing of the Mandelbrot set. In the picture 5 there is presented time of computations with the change of the number

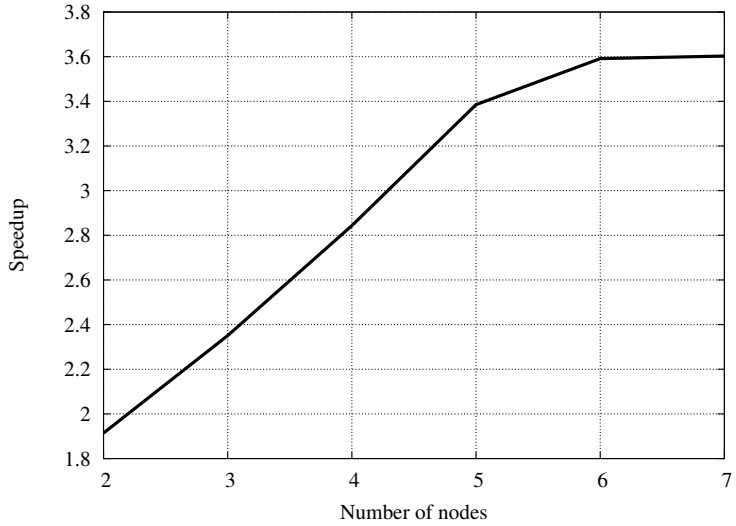

Figure 7. Speedup - maximum with different number of subtasks for every number of nodes.

of tasks for particular number of nodes. In the picture 6 we have presented change of the computation time with the change of the number of nodes for fixed number of subtasks. Finally in the picture 7 there is presented speedup of system. The results show that UNICORE Desktop grid works efficiently for small number of nodes and saturation is achieved when size of the task is small.

6 Conclusions and future work

Our work proofs that it is possible to built desktop grid based on the UNICORE middleware. Some additional components such as desktop grid manager and https proxy were missing and have been developed here. We have performed efficiency tests which proof possibility of building desktop grid infrastructure of several and even hundreds of computers.

The scalability tests with larger number of nodes still have to be performed, but built infrastructure seems to be limited mostly by the scalability of the UNICORE components.

Another significant issue is development of the scalable applications for the particular middleware. Hopefully, this can be done in the form of the Java class which can be easily developed.

Presented work shows interesting and to some extend not predictable before application of the UNICORE middleware.

Acknowledgements

This work has been supported by the KARDIONET project.

References

1. Anurag Acharya, Guy Edjlali, and Joel Saltz. The utility of exploiting idle workstations for parallel computation. *SIGMETRICS Perform. Eval. Rev.*, 25(1):225–234, 1997.
2. Matt W. Mutka. Estimating capacity for sharing in a privately owned workstation environment. *IEEE Trans. Softw. Eng.*, 18(4):319–328, 1992.
3. David A. Patterson, David E. Culler, and Thomas E. Anderson. A case for now (networks of workstation). In *PODC '95: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, page 17, New York, NY, USA, 1995. ACM.
4. David P. Anderson. Boinc: A system for public-resource computing and storage. In *5th IEEE/ACM International Workshop on Grid Computing*, pages 4–10, 2004.
5. Nabil Abdennadher and Regis Boesch. Towards a peer-to-peer platform for high performance computing. *High Performance Computing and Grid in Asia Pacific Region, International Conference on*, 0:354–361, 2005.
6. Attila Csaba Marosi, Gábor Gombás, Zoltán Balaton, Péter Kacsuk, and Tamás Kiss. Sztaki desktop grid: Building a scalable, secure platform for desktop grid computing. In *Making Grids Work Proceedings of the CoreGRID Workshop on Programming Models Grid and P2P System Architecture Grid Systems, Tools and Environments 12-13 June 2007*, Heraklion, Crete, Greece, 2007.
7. Zoltan Balaton, Zoltan Farkas, Gabor Gombas, Peter K. Kacsuk, Robert Lovas, Attila Csaba Marosi, Ed Emmen, Gabor Terstyanszky, Tamas Kiss, Ian Kelley, Ian Taylor, and Filipe Araujo. Edges, the common boundary between service and desktop grids. *Parallel Processing Letters*, 18(3):433–445, September 2008.

DEISA Development Environment

Giuseppe Fiameni[1], Mohammad Shahbaz Memon[2], Siew Hoon Leong[3], Xavier Pegenaute[4], Judit Jimenez[4], Claudio Gheller[1], Riccardo Brunino[1], and Daniel Mallmann[2]

[1] CINECA - Consorzio Interuniversitario, Bologna, Italy
E-mail: {g.fiameni, c.gheller, r.brunino}@cineca.it

[2] Jülich Supercomputing Centre - Research Centre Jülich, Germany
E-mail: {m.memon, d.mallmann}@fz-juelich.de

[3] LRZ - Leibniz-Rechenzentrum, München, Germany
E-mail: siew-hoon.leong@lrz.de

[4] BSC - Barcelona Supercomputing center, Barcelona, Spain
E-mail: {judit, xavier.pegenaute}@bsc.es

In order to fully exploit the DEISA[a] supercomputing infrastructure, effective software development tools, customized over resource specifics, must be provided to the user communities in order to facilitate the process of developing scientific applications. The goal of our work is to provide an effective environment for application development, based on a software platform where different tools coexist and work together. Furthermore, we aim to provide a common user interface to operate across different computing platforms, while remaining agnostic to the actual tools deployed on the back-ends.

1 Introduction

In recent years, it has become evident that high-performance application development must be supported by effective tools in order to achieve the goal of a sustained utilization of high-performance systems. Users perceive a lack of tools on the ever increasingly complex high performance computing system as a frequent impediment that keeps them from realizing the full potential of the machines they are accessing. However, many tools are actually available but they are either only supported on a limited number of platforms or have to be adapted to the infrastructure before being employed easily or effectively in the application development process. This work proposes a software infrastructure where a number of tools, necessary for application development on high performance computing system, can coexist and operate together to provide a common user interface across heterogeneous computing platforms like DEISA. The software infrastructure is based on Eclipse[5] and on the Parallel Tools Platform (PTP)[1]. Eclipse is a well-known Java-based integrated development environment (IDE) that provides numerous tools, all accessible from a common graphical interface, which are essential for developing enterprise level applications. The integration between Eclipse and PTP, encompassing other components, will bring to researchers and software developers a standardized and uniform interface

[a] DEISA *Distributed European Infrastructure for Supercomputing Applications* is a consortium of leading national Supercomputing centers that aims at fostering the pan-European world-leading computational science research. Deploying and operating a persistent, production quality, distributed supercomputing environment, it aims at delivering an operational solution for a future European HPC ecosystem.

to effectively and efficiently perform all the steps which are necessary to develop and execute parallel applications across heterogeneous resources. The resulting environment is referred as the DEISA Development Environment (D2E). The functionalities provided by the environment have been tailored to DEISA infrastructural requirements and specifics. Missing DEISA key-functionalities have been implemented; these include the support of the DEISA compliant authentication method based on the GSI-SSH protocol[3], the support of the module system to automatically set up environment configuration, the support of profiling tool (i.e. PARAVER[6]) and the integration of UNICORE[9] Rich Client tool to support easy job submission.

Using such an integrated environment, application developers will immediately be able to take advantage of the following:

- a customizable environment that permits user to personalize the environment interfaces and perspectives;
- a simplified approach for porting applications across heterogeneous execution environments;
- a dynamic platform that can be easily extended to accommodate new tools with the consequence to provide new functionalities;
- a customizable environment that permits user personalize the environment interfaces and perspectives as they wish.

The aim of this paper is to present a part of the work performed in realizing the entire D2E platform. More precisely, it will focus on the integration of the UNICORE Rich Client with the existing PTP components for managing the execution of computational jobs.

2 Architecture

Fig. 1 shows an overview of the system architecture, including the relationships that exist among the high-level actions (edit, compile, debug, execute, profile) and the system components. As shown in the picture, the actions rely on different server side components, each carrying out the operations it provides. Each DEISA site "sees" the shared GPFS[b] file-system and offers specific computing resources. Each of them runs a PTP Proxy, a hub, where commands from the clients are interpreted and executed on the remote system. The execution of computational jobs, based on the "batch" approach, is managed by the UNICORE plug-in.

It is worth mentioning that the Eclipse-PTP platform comes with most of the components necessary to support remote operations. However, some of them had to be modified or improved to overcome existing problems and/or to implement new features that are indispensable for interoperability with the DEISA infrastructure. Examples are adoptions of UNICORE for job submission and GSI-SSh as an authentication mechanism.

[b]The General Parallel File System *GPFS* is a high-performance shared-disk clustered file system developed by IBM.

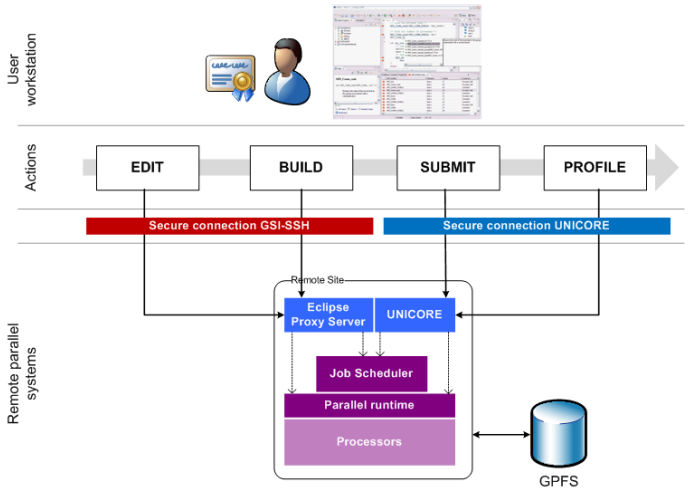

Figure 1. Schematic picture of the D2E architecture.

3 D2E plug-ins for PTP

Eclipse is an extensible platform either for creating specific IDEs or developing stand-alone applications. It provides a set of core services to manage high level tools that work together to support code programming and execution of application specific tasks. Developers may contribute to the Eclipse platform by wrapping their tools in pluggable components, called Eclipse plug-ins, conforming to Eclipse's plug-in contract. The basic mechanism of extensibility in Eclipse is that new plug-ins can add new functionalities to existing ones.

3.1 Job Execution: UNICORE-PTP

The UNICORE-PTP realization is a set of plug-ins implemented to integrate UNICORE with the Eclipse PTP. Besides the PTP plug-ins, UNICORE-PTP imported several components from the existing UNICORE's native Rich Client Platform (RCP) code base. The RCP plug-ins inherently provides the functionality of UNICORE middleware related client side features such as remote stubs and helper APIs to run typical grid use cases. Fig. 3 shows the two tiered PTP architecture where the upper layer represents the Eclipse Client, which contains the Runtime Views, Runtime Model, and Runtime Controller.
The Eclipse Client tier in Fig. 3 also shows the presence of UNICORE elements on multiple of its sub-components. The integration design does not augment any of the existing PTP plug-in sources, thus we employed the standard Eclipse based declarative plug-in extension mechanism. The second tier is a resource manager layer representing multiple types of resource managers, batch systems, and UNICORE instances. On this layer UNICORE does not need any functional changes at the source code level. The only administrative configuration required on the server side is to declare MPI application with respective paths. This configuration is tested against the UNCORE's production version deployed under the DEISA infrastructure. As UNICORE is not providing a debugging feature and any inter-

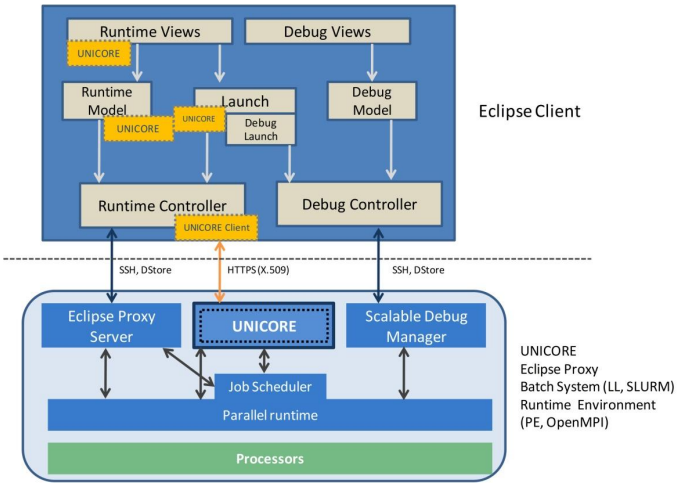

Figure 2. PTP Architecture with UNICORE components.

action with the PTP based server proxies, therefore no extension has been implemented to communicate with the Eclipse Proxy Server, and Debug components.

3.2 Runtime Views

In order to offer users with GUI elements that will help them using UNICORE functionality within PTP landscape, we have defined some extensions as plug-ins to PTP's Runtime views. For clients targeting UNICORE server side instances, these plug-ins have been implemented via extending the resource manager creation/update and the job launch configuration wizards.

3.3 Resource Manager Wizards

The Resource Manager wizards appear when users intend to create or update resource manager instances. PTP offers a framework to add non-grid enabled resource manager components, but it does not support grid middleware such as UNICORE. In this work package we have developed the set of extensions and new components for integration. By using the PTP abstract wizard interfaces the UNICORE specific wizard pages are created. This will allow users to easily manage UNICORE instances within PTP Runtime Perspective. The second step for the user creating a resource manager instance is to specify the UNICORE related information. For mounting a UNICORE instance user has to provide Registry URL which points to the remotely deployed Registry service which contains the addresses of the available UNICORE managed target systems and storage management services. As soon as user enters the URL and confirms it, the client validates the address and retrieves the available target system entries. In case of supplying the wrong Registry URL, the client GUI will not allow user to accomplish the Resource Manager creation phase. After having successfully created the UNICORE instance, the information related to Registry and target systems are populated to Runtime Model (see section 3.5).

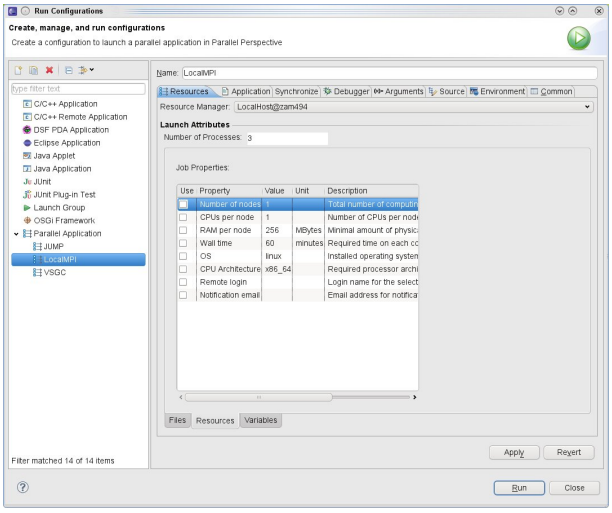

Figure 3. Parallel Application Launch configuration showing UNICORE specific Resources Tab to capture compute resources requirements.

3.4 Launch Configuration Extension

In order to attain UNICORE specific launch within PTP framework, this extension point overrides the PTP's Resource launch tab. Fig. 3 shows the new UI forms where user can easily specify UNICORE oriented file staging information, resource requirements, and environment variables. It is also important to mention that D2E only extended the PTP's Resources tab. Apart from the Resources tab, the PTP framework comes with the Application, Debugging, and Environment configuration tabs, and one can easily argue that why these other tabs are not subjected for extension. Although the UNICORE-PTP requires some tabs to be remove or altered, but due to the current limitation of PTP launch framework and its static nature, it is not possible to embed UNICORE elements within all of the already provided launch configuration tabs.

3.5 Runtime Model

The runtime model corresponds to the hierarchical representation of resources and jobs during the parallel application development lifecycle. In this lifecycle, for running an application on real resource manager or runtime environment, UNICORE is taken as a role of PTP enabled logical Resource Manager, providing abstraction of multiple physical batch systems and runtime environments. In fact, UNICORE is a grid middleware, while PTP can only represent the bare resource management system functionalities. Therefore, PTP does not effectively cater model representation requirements of grid systems. In order to deal with this, we have defined some additional attributes to support single and multiple UNICORE sites in order to realize infrastructure information. In UNICORE-PTP main attributes are related to Registry and target systems required for managing job execution and monitoring functions. The Registry plays a vital role in the Runtime model, therein

every UNICORE instance within the PTP ecosystem stores its information in order to realize the resource and job monitoring use cases. Additionally, the job related entities were also introduced such as job state, resource requirements, job addressing information, list of staging-in entries, job download directory, error information, local/remote project association, and launch configuration model.

3.6 Runtime Controller

Runtime Controller component contains a proxy or client side stubs. The proxy or client stubs are functional components responsible for preparing requests to remote resource managers. In PTP perspective, the Runtime Controller (see Fig. 2) architecturally sits at the lower level to interact with remote RMS proxies by sending remote commands. In our case, no remote proxy is required, the client side wrappers that call out to remote web service endpoints are UNICORE client libraries acting as runtime controllers, and on the server side it is assumed that the UNICORE installation is already running. For integration purposes, client side components extend the PTP's Runtime Controller to contact remote UNICORE sites, prepare job requests, submit and monitor jobs using launch configuration information, and download job outcomes when a job is finished. It is also worth mentioning that on the transport level PTP uses SSH or DStore protocol, and UNICORE-PTP uses HTTP(S) to contact remote UNICORE sites.

4 Concluding Remarks

This paper has presented the D2E platform focusing on the integration of the UNICORE Grid System as resource manager for PTP. The benefits of adopting the D2E for the development of scientific applications have been reported. The adoption of the platform in production environment looks promising even though some instabilities problems need to be addressed before distributing the software to the user community. The first release of the D2E bundle can be downloaded from the DEISA repository[4] site as a self-consistent archive.

References

1. PTP *Parallel Tools Platform*. `http://www.eclipse.org/ptp/`
2. DEISA *Distributed European Infrastructure*. `http://www.deisa.eu`
3. GSI-OpenSSH documentation. `http://globus.org/toolkit/docs/latest-stable/security/openssh/`
4. DEISA Code Repository. `http://work.deisa.eu/svn/WP8/bundle`
5. ECLIPSE. `http://www.eclipse.org`
6. PARAVER. `http://www.bsc.es/paraver`
7. Greg Watson, Nathan Debardeleben *Developing Scientific Applications Using Eclipse* (Computing in Science & Engineering magazine, July/August 2006).
8. Greg Watson, Craig Rasmussen *A Strategy for Addressing the Needs of Advanced Scientific Computing Using Eclipse as a Parallel Tools Platform*, White Paper, LA-UR-05-9114, December 2005.

9. A. Streit et al *UNICORE 6 - A European Grid Technology*, High Speed and Large Scale Scientific Computing, ed.: L. Grandinetti, G. Joubert, W. Gentzsch, Amsterdam, IOS Press, 2010, Advances in Parallel Computing Vol. 18. - 978-1-60750-073-5. - S. 157 - 173

Secure High-Throughput Computing Using UNICORE XML Spaces

Richard Grunzke[1] **and Bernd Schuller**[2]

[1] Center for Information Services and High Performance Computing
Department of Distributed and Data intensive computing
Dresden University of Technology, 01062 Dresden, Germany
E-mail: richard.grunzke@tu-dresden.de

[2] Jülich Supercomputing Centre,
Distributed Systems and Grid Computing Division
Forschungszentrum Jülich GmbH, 52425 Jülich, Germany
E-mail: b.schuller@fz-juelich.de

In this work we are concerned with running a large compute cluster in "farming" mode, i.e. when the typical job uses one core and the number of jobs running at the same time ideally is equal to the number of cores. The number of cores is of the order of 100000. Using Grid middleware, specifically UNICORE, to efficiently run large numbers of jobs on such a compute clusters brings a number of challenges. Currently, UNICORE 6 is well suited for highly parallel HPC jobs, but does not scale well to very high numbers of jobs. In the space-based approach that has been already described in a previous paper[3], an XML document repository (the Space) is used as a central job queue. Clients submit jobs into the space, and the execution server(s) pull jobs from the space for processing. Since no further interaction of the client is required, the job throughput is improved. Since the execution server controls the number of concurrent jobs, its efficiency is also improved. The prototype described in[3] has recently been extended to include trust delegation, making the system capable of multi-user operation. In this paper, we explore the performance characteristics of the space-based approach and compare it with the standard batch mode of the UNICORE commandline client. Issues such as throughput, scalability, stability and efficiency (including client side CPU usage) are investigated, and the suitability of the different approaches for various usage scenarios is evaluated. The aim of this work is to help bring UNICORE 6 to the next level of high throughput computing.

1 High-Throughput Computing

The cooperation between the Center for Information Services and High-Performance Computing (ZIH) and the Max Planck Institute of Molecular Cell Biology and Genetics (MPI-CBG) consists of handling millions of images for automatic image analysis. Biologists at the MPI-CBG are conducting experiments in the area of molecular mechanisms of endocytosis and endosome biogenesis. In the experiments they illuminate parts of the examined cells to investigate the cellular processes. Automatic microscopes then create images of different stages of the processes inside the cells. Since many images are created, the analysis exceeds the computational capabilities of MPI-CBG. The high-performance computing systems at ZIH are used to support the scientists in analyzing the images. To optimally support this high throughput scenario, the middleware has to be highly efficient for the imposed overhead to be as low as possible.

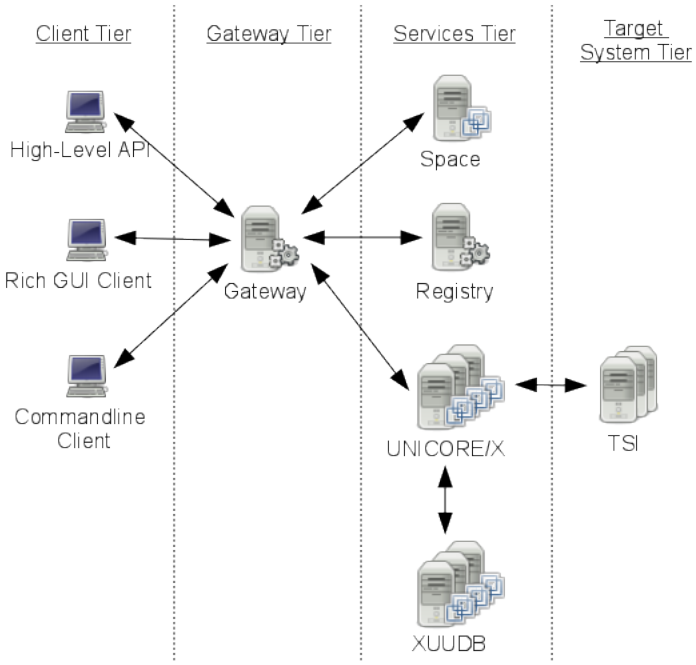

Figure 1. UNICORE 6 Architecture

2 High-Throughput Computing using UNICORE

2.1 UNICORE

UNICORE, developed in the course of several German and European projects since 1997[1], is a mature Grid middleware that is deployed and used in a variety of settings, from small projects to large (multi-site) infrastructures involving high-performance computing resources. UNICORE can be characterised as a vertically integrated Grid system, that comprises the full software stack from clients to various server components down to the components for accessing the actual compute or data resources. Its basic principles are abstraction of site-specific details, openness, interoperability, operating system independence, security, and autonomy of resource providers. In addition, the software is easy to install, configure and administrate. The latest version is UNICORE 6, which is based on Web Services and particularly the Web Service Resource Framework (WSRF). UNICORE is licensed under the liberal BSD license, and is available as open source from the SourceForge repository[2].

As shown in Figure 1, UNICORE 6 is a four-tiered system, consisting of the client, gateway, services and target system tiers. A wide variety of clients exist, from programming APIs[4], a commandline client[5], to a rich client based on the Eclipse framework. The Gateway is a thin authentication and routing service that can be considered as a web service firewall and router. It resides outside the networking firewall, protecting the services behind it. Thus, UNICORE by default only requires a single open port to the public inter-

net. The basic services (UNICORE atomic services) provide resource discovery (Registry service), job execution (Target System Factory and Target System services), and file access (Storage and FileTransfer services). The target system tier consists of the interface to the local operating system, file system and resource management (batch) system.

UNICORE 6 uses XML based standards wherever possible. The basic communication and services integration is based on SOAP web services and the web service resource framework (WSRF). The OGF standard job submission description language (JSDL) is used for job submission. In the security domain, SAML assertions are used for trust delegation, and XACML policies control the authorisation of web service calls.

For high throughput computing, the UNICORE commandline client UCC is used. UCC offers a number of commands for interacting with a UNICORE Grid.

2.2 UCC Run Mode

The UCC *run* command is used to run a single job, which can of course be done for example in a shell script. However, the run mode means that UCC is started once for every job, which results in a huge overhead for every job due to the time taken for startup of the Java virtual machine (VM), and checking registry connection and target system availability. This explains the poor performance of about 0.4 jobs/s in Figure 2. The measurements in Figure 2 were done with submitting 1000 jobs each and taking the time it took for all jobs to finish. The UCC run mode is only advised for a few number of long jobs, because then the overhead is not significant.

2.3 UCC Batch Mode

To alleviate the overheads mentioned above, a *batch* command is available in UCC. In this mode, UCC is only started once and it continuously reads job description files and submits them as jobs to the UNICORE 6 middleware. As can be seen in Figure 2 the batch mode works OK for high throughput scenarios with about 2.2 jobs/s. This approach is useful for a medium number of jobs and is widely available since it just needs a standard installation of the UNICORE 6 middleware. The following shortcomings severely limit the usability of the approach with a very high number of jobs.

- It causes a high CPU utilization on the client side, a quad core workstation was fully loaded during the measurements in Figure 2.
- It is hard to make efficient use of the UNICORE/X, with it running stable at the same time. The client has to decide where to send the jobs. For efficient scheduling it needs detailed load information about every UNICORE/X.
- This approach does not scale with the number of resources because the client has to be aware of an increasing number of UNICORE/X and communicate with them.
- The communication between the UCC and UNICORE/X is coupled, meaning that for every job the following happens; Finding of an appropriate site, submitting the job and polling in regular intervals for the status of the job.

A more efficient, scalable and decoupled approach is needed to optimally support a very high number of jobs.

3 Space Based Approach

As described in a previous paper[3], an extension component for UNICORE, the so-called Space, allows to store and retrieve XML documents. Documents can be read and removed from the space using a XML document template to specify the requested document. In a high throughput scenario, this XML document repository can be used as a central job queue. Clients submit jobs to the space, and execution sites can take jobs for processing. This approach offers several advantages. Firstly, the client need only communicate with the space, independent of the number of execution sites. This greatly reduces the client-side communication overhead. Secondly, the execution sites can process jobs at their own pace, and taking their own local policies into account. For example, it is trivially easy to let a site only process jobs meeting certain criteria, for example where a minimum or maximum number of CPUs is requested.

A UCC extension is also available which offers a generic *space* command for storing and retrieving XML documents. A special *space-batch* command has also been added that deals with special job documents. On the UNICORE/X server side, a *job taker* component reads jobs from the space and submits them to the UNICORE execution manager. Multiple job takers can be deployed per site.

For realistic multi-user scenarios, the Space had to be extended with an (optional) access control mechanism (ACL), which prevents users from removing other user's documents from the space.

4 Benchmarks

The following section is dedicated to the investigation of the space based approach during runtime. First the number of client threads is investigated to see how that changes the performance. Then the updateperiod parameter, ACLs, number of jobtaker and value of bunchsize is varied to find implications on the performance.

Figure 3 shows the number of threads on the client side used for submitting 1000 jobs to the space server. The measurement shows that when this parameter is 2 or above it does not has a significant impact on the performance.

Figure 4 shows an investigation of the different updateperiod values. This parameter defines how often the jobtaker looks for new jobs in the space. The measurement shows that if the value is too high, the performance is decreasing. It also shouldn't be too low (below 1 second) to not cause unnecessary load on the space server.

Figure 5 shows the comparison with ACL checking on and off. With ACL checking on it is only slightly slower, thus it is advised to turn ACL checking on.

Figure 6 presents an interesting result of a measurement which compares handling 1000 jobs with 2 jobtaker on 1 UNICORE/X, 4 jobtaker on 1 UNICORE/X and 4 jobtaker on 2 UNICORE/X. The throughput decreases by 20% when the number of jobtaker on one UNICORE/X are doubled from 2 to 4. When now the 4 jobtaker are distributed over 2 instead of 1 UNICORE/X the performance almost doubles. This shows that jobtaker on

one UNICORE/X block each other. Here lies the potential to significantly improve the performance.

The bunchsize specifies how many jobs a jobtaker fetches form the space in one run. Figure 7 shows values from 1 to 10. The throughput is increasing and starting from a value of 20 in figure 8 the throughput stays roughly the same. From 100 on in figure 9 the performance is slightly decreasing. It is advised to use a bunchsize between 20 and 100 for optimal performance.

5 Conclusions

The space based approach is highly capable of handling a large number of jobs. It is highly scalable since new UNICORE/X services can be added in a transparent way for the clients and the space service. Furthermore the client is not the bottleneck anymore, since while using space-batch the client does not cause significant CPU load. UNICORE/X is now optimally configurable for stability and efficiency, various parameters can now be fine tuned. A drawback is that direct file uploads from the client are not possible, input needs to be uploaded to a storage instance and staged-in from there to the uspace. The throughput of 3 jobs/s is already fairly good, and is likely to be improved due to optimizations in regard to running multiple jobtaker on one UNICORE/X. Further tests to find bottlenecks and limitations are planned to find further possibilites to improve the performance. Also multiple space support is planned.

References

1. UNICORE. `http://www.unicore.eu`
2. UNICORE SourceForge project.
   `http://sourceforge.net/projects/unicore`
3. Schuller, B.; Schumacher, M.,
   Space-Based Approach to High-Throughput Computations in UNICORE 6 Grids
   in: Proceedings of 4th UNICORE Summit 2008 in Springer LNCS 5415,
   Euro-Par 2008 Workshops: Parallel Processing, pp 75 - 83.
4. Menday, R.; Hagemeier, B., HiLA 1.0.
   `http://www.unicore.eu/community/development/hila-reference.pdf`
5. UNICORE commandline client.
   `http://www.unicore.eu/documentation/unicore6/manuals/ucc`

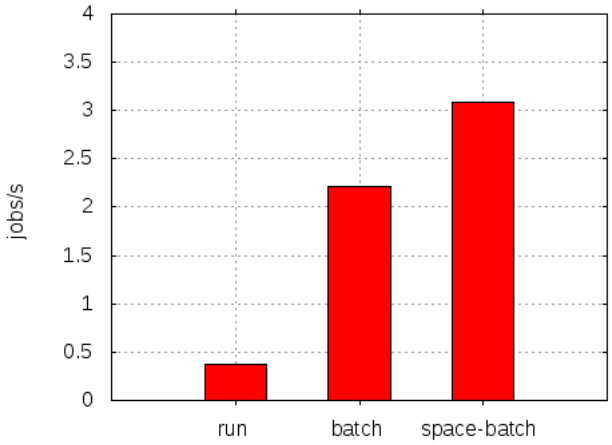

Figure 2. Comparison of the ucc "run", "batch" and "space-batch" mode

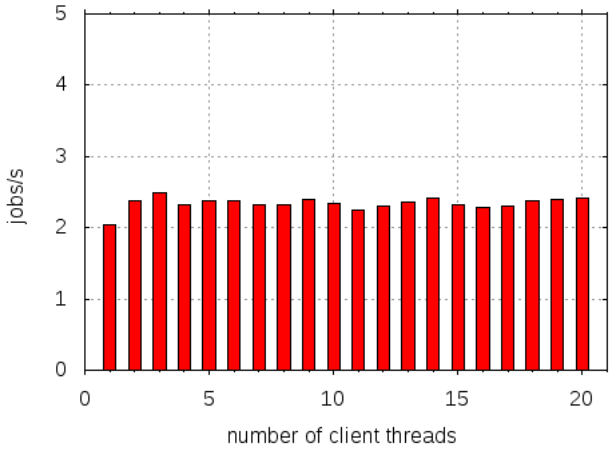

Figure 3. Comparison of different numbers of client threads

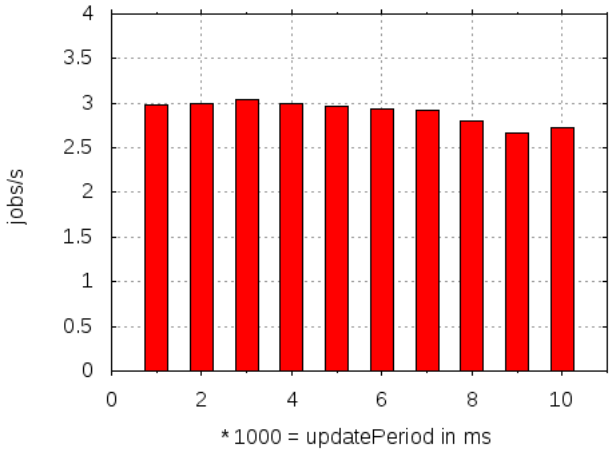

Figure 4. Different numbers of updateperiod values

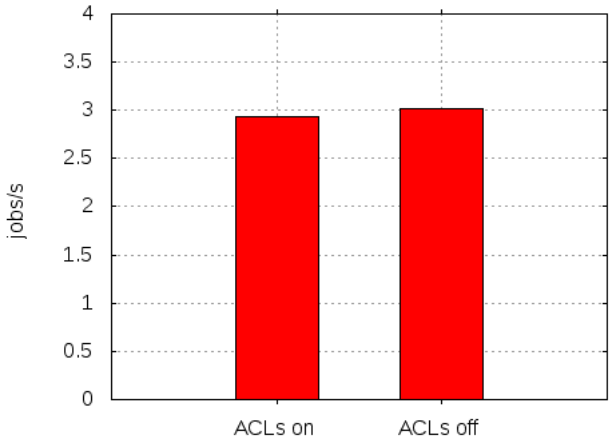

Figure 5. Comparison with ACLs on and off

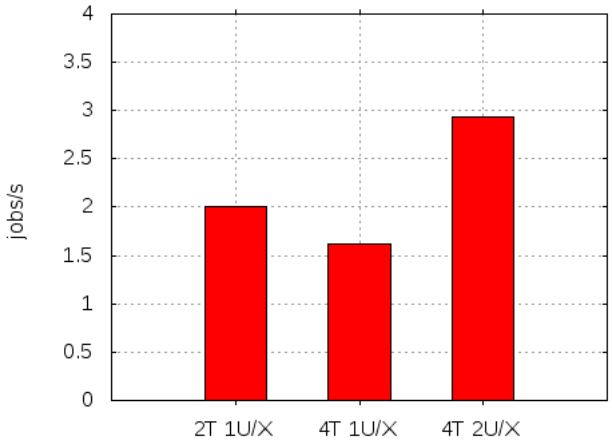

Figure 6. Comparison of combinations of 2 and 4 jobtaker and 1 and 2 UNICORE/X

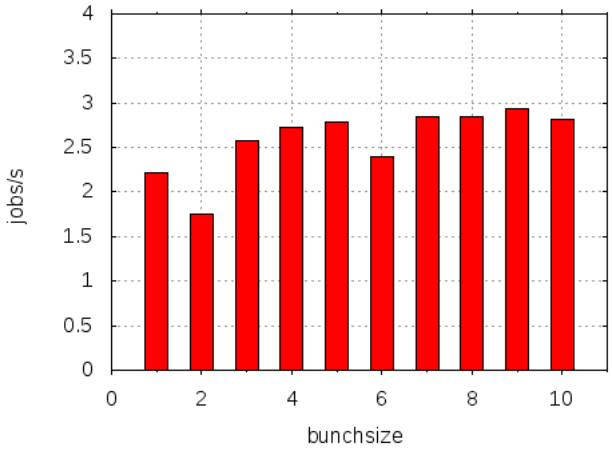

Figure 7. Comparison of bunchsize values from 1 to 10

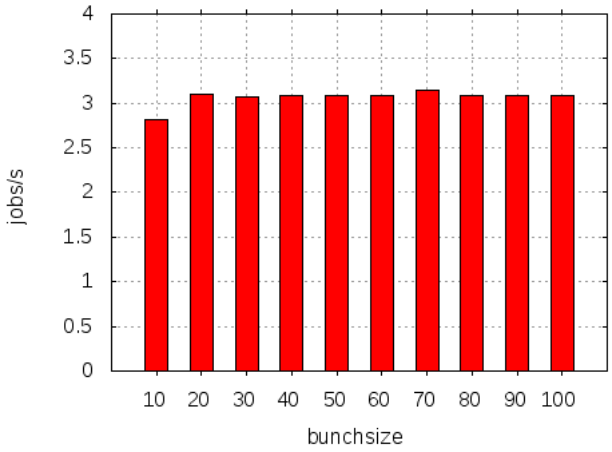

Figure 8. Comparison of bunchsize values from 10 to 100

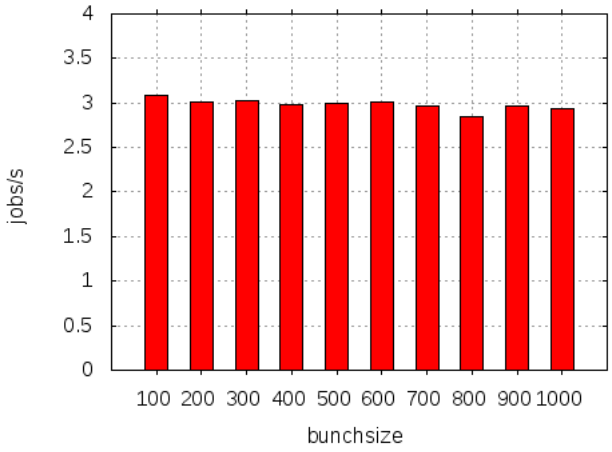

Figure 9. Comparison of bunchsize values from 100 to 1000

Synergizing ETICS and UNICORE Software

André Giesler[1], Achim Streit[1], Elisabetta Ronchieri[2], and Michele Dibenedetto[2]

[1] Jülich Supercomputing Centre,
Research Centre Jülich, 52425 Jülich, Germany
E-mail: {a.giesler, a.streit}@fz-juelich.de

[2] INFN-CNAF, Viale Berti Pichat 6/2,
I-40127 Bologna, Italy
E-mail: {michele.dibenedetto,elisabetta.ronchieri}@cnaf.infn.it

ETICS 2, a collaborative project funded by the European Commission, has produced a number of developments improving the ETICS system itself and UNICORE 6 mutually. In this paper, we present the two most significant results of this collaboration. It is shown how the whole UNICORE 6 software stack has been integrated in the ETICS environment to enable a quality assured building and testing of the UNICORE software modules across different platforms. Furthermore, we explain the extending of the ETICS system by developing an UNICORE-based job-submission service, so that ETICS build and test jobs can be managed by the UNICORE 6 Grid middleware.

1 Introduction

A couple of commercial partners and research institutes collaborated in the ETICS 2[1] project which could both contribute their development experience and proven software concepts from the Grid-computing environment. In particular, this concerned the already existing ETICS software itself as well as the Grid middleware UNICORE 6[2], which is especially evolved by the ETICS 2 partner Forschungszentrum Jülich. Both software products have strengths and capabilities from which they could benefit mutually. It appeared that the advantages of the ETICS environment are especially in its capabilities to integrate arbitrary software repositories, build-tools, and programming languages. Furthermore, ETICS guarantees a software building and testing quality process on different software architectures and platforms. This flexibility allows UNICORE 6 software modules to be build in the ETICS system and concurrently make use of all benefits concerning the provided hardware-and-software heterogeneity.

On the other hand, ETICS is anxious to open up its infrastructure to new user groups and fields of application. This includes especially High Performance Computing (HPC) users and applications. In this area users are often familiar with HPC Grid middleware like UNICORE 6. So far, ETICS trusted a dedicated Condor-Metronome solution[10] for submitting its build and test jobs to its infrastructure. However, the Metronome software is not available in most of HPC environments, representing clearly a barrier for bringing ETICS to this area. For that reason it was necessary to re-design the ETICS job submission service to integrate potential submission engines from the HPC world such as UNICORE 6. We describe in this document the efforts made to create a new Submission Web Service to let ETICS benefit from the capabilities of a Grid middleware like UNICORE 6.

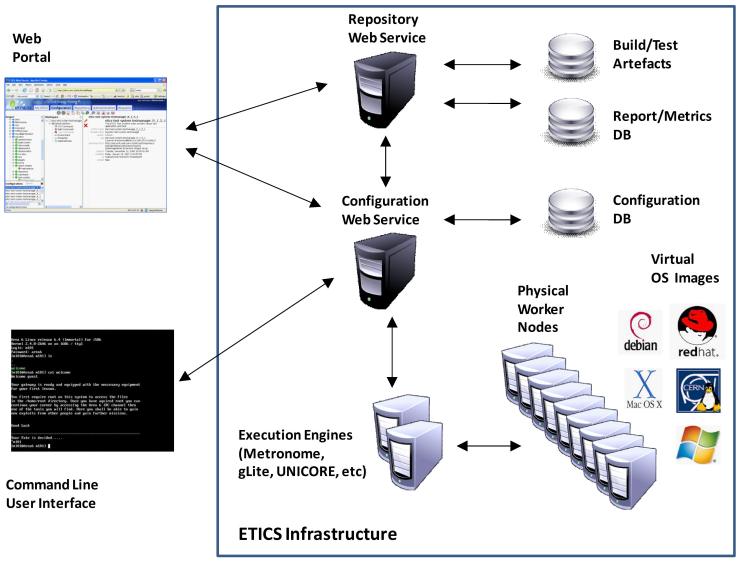

Figure 1. System overview of the ETICS System .

These ETICS system will be outlined in Sec. 2. Synergizing benefits for UNICORE in the collaboration with ETICS are discussed in Sec. 3. The benefits for ETICS, more precisely integrating the UNICORE 6 middleware in the ETICS resource management system, are described in Sec. 4.

2 ETICS

ETICS stands for eInfrastructure for Testing, Integration and Configuration of Software. The ETICS system can be described as a build and test framework for arbitrary software developments. It was designed starting from existing open-source software, smartly integrated and adapted according to the requirements of the software projects using ETICS. The system is able to manage small and large software projects written in any programming languages. It also supports software professionals by providing shared computing resources (i.e., platforms according to the ETICS terminology), and by automating the execution of their builds, tests and software quality verification activities reducing in this way the burdens of manual processes. Furthermore, the possibility of sharing the build and test infrastructure between different software projects reduces software project maintenance costs. Fig. 1 illustrates an overview of the basic architecture of the ETICS system.

The ETICS system provides a set of tools, services, and repositories that can be used to run builds and tests on users' computers, but also to submit remote builds and tests on multiple platforms at the same time. For this purpose the ETICS infrastructure offers a pool of platforms, representing particular combinations of operative system types, architectures and compilers. To increase the number and type of platforms provided by ETICS, an easy

solution is given by the integration of the ETICS infrastructure into existing Grids and distributed computing infrastructures, such as Amazon Web Service[4] (EC2, SQS, SimpleDB and S3), or EGEE[3] (Enabling Grids for EsciencE), gathering together (tens of) thousands of computing resources and optimizing their usage. ETICS uses source code management systems, like CVS or subversion , and can be easily extended with support for more. It provides a rich library of quality assurance tools, such as checkstyle, junit, or cppunit, which projects can activate as they wish when running builds and tests. ETICS users can choose between a web based portal client and a command line client to create configurations of their software or performing remote builds or tests.

3 Building UNICORE Components with ETICS

The UNICORE 6 development is centrally operated at the SourceForge[5] source code repository. SourceForge is a web-based source code repository and acts as a centralised location for software developers to control and manage their software development. Furthermore, UNICORE 6 is using project management and build automation tools like Apache Maven or Apache Ant, since the UNICORE source code is written almost entirely in the Java programming language. Finally, the UNICORE software version controlling is managed by the revision control system Apache Subversion. So, it is mandatory that an environment, which controls the UNICORE 6 software building process, is able to support all of these state-of-the-art tools like Maven, Ant, or Subversion. The ETICS system provides support for all these tools. Thus, it is guaranteed that all common UNICORE 6 development processes can be managed within that infrastructure.

In principle, the UNICORE development will stay at Sourceforge for the moment. However, it is nevertheless advantageous to UNICORE to use also ETICS for development, which means to build some of its software modules in ETICS. The first benefit is to build the UNICORE components on different hardware-and-software architectures which are provided and managed by the ETICS infrastructure. The developers do not need to configure their own testing pool with all the potential architectures where UNICORE software could be deployed. They simply create an appropriate ETICS configuration for the desired platform and test the remote build of the software component there. Another benefit comes from the ETICS A-QCM[6] plug-in. This feature enables an automatic quality verification and certification of the software which is build with ETICS. The current ETICS A-QCM plug-in focuses on the static analysis of classes and code, the structural tests, the functional tests, the standards and operating system compliance. Thereby, UNICORE 6 can automate the quality certification activities for its components. Furthermore, the development progress of UNICORE can be monitored in order to maintain a good level of quality during the production lifecycle.

The most practical benefit for UNICORE comes from the ETICS functionality to create end-user packages. This feature supports automatic creation of distribution packages in a number of different formats (rpm, deb, tgz) on the basis of the platform selected for the software build process. Since many users want to have pre-built UNICORE modules for individual Linux platforms, it is very convenient for UNICORE developers to create the requested components in an automatic manner. ETICS provides an in-built packaging

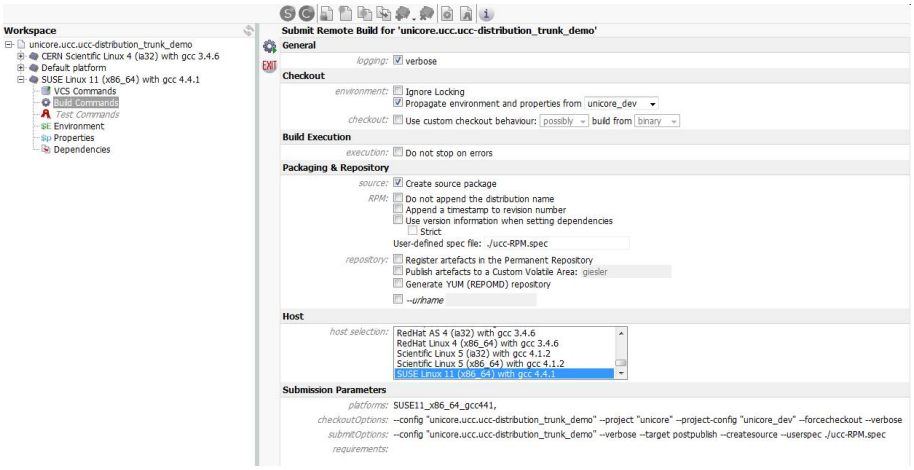

Figure 2. ETICS remote build creates a RPM package of UNICORE UCC module.

feature as well as the alternative option to specify an individual user-defined packaging behaviour. The latter option comes towards UNICORE developers, since they want to distribute the software files in predefined directories which can be achieved by specifying a customized package configuration file, e.g. a RPM spec file.

Fig. 2 shows the panel in the ETICS Web client where the remote build to create an RPM package of the UNICORE module, in this case the UNICORE Command line Client (UCC) is defined. In the workspace panel on the left hand of the figure is an ETICS configuration defined, which controls the remote build of the UCC module on a SuSE Linux machine. The ETICS configuration consists amongst others of commands to checkout the module from SourceForge SVN repository, defining third party dependencies like correct Java versions, or setting the right Maven commands. On the right hand of the figure special parameters related to the remote build itself can be given. For instance, the ETICS built-in RPM packaging mechanism is overwritten by setting a user defined RPM-spec-file.

4 Extending the ETICS system by an UNICORE-based Submission Service

The ETICS execution service is responsible for the execution of remote build and test jobs submitted by ETICS users. Originally, this service only relied on the NMI Metronome[10] software. However, to allow distributed computing infrastructures (i.e., EGEE and DEISA) to use their own job management systems, the ETICS submission service has to be able to perform job operations with the different standard-middlewares of these infrastructures. It is obvious that environments like EGEE or DEISA are not willing to have the overhead of installing Metronome, and move parts of their resources out of their infrastructures to the Metronome Pool. Therefore, the resource management logic of ETICS was re-engineered during ETICS 2 to allow the integration of arbitrary job engines

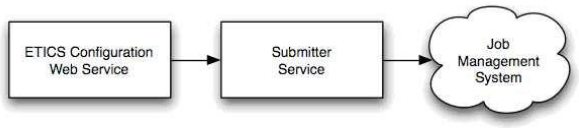

Figure 3. Extended ETICS submission architecture.

in the system. As a result the new standalone ETICS Submission Web Service (ESWS) was designed providing a generic submission interface Fig. 3.

The logic of the ESWS is hidden inside the architecture so that an ETICS user only need to submit a build or test job with the standard ETICS clients. The ESWS is based on a pluggable Web Service interface that contains operations to submit the build or test job, to query the status of an ongoing build, to delete submitted jobs, and upon completion to upload the results of the build or test to the ETICS repository for further processing. Plug-ins to this ESWS framework are defined by implementing a few interfaces, corresponding to the required ESWS operations.

The ESWS architecture (Fig. 4) has been designed to keep a clear separation among the basic Grid job operations, so that the logic of different middleware can be easily integrated within the ETICS framework. The service is composed of several sub-services that reside either in the main execution thread of the Web servlet or in dedicated threads with periodical execution. The Unique ID Generator service provides a way to obtain a unique identifier, called Global ID, in the form of a Universally Unique Identifier (UUID) that is unique for each invocation of the service. The ESWS uses the identifiers to uniquely identify each reiceived submission request. The mapping between the Global Id and job identifier generated by the plugged-in middleware is managed by the ID Mapping Service, which maintains the information in a persisted database for recovering it in case of any issues. The Status Fetcher service is responsible for retrieving the output of each submission once the submitted job has completed its execution. For this purpose the Status Fetcher service periodically queries the status of each job that has not yet completed its execution process. Since ETICS job statuses and middleware job statuses are not inconclusively the same, a Status Mapper service was implemented to translate the middleware specific strings to a common internal ESWS representation. Finally, the Result Uploader Service is responsible for uploading the finished software packages and build reports to the ETICS repository as well as for informing the user about the result of the job.

4.1 The UNICORE Submitter plug-in

So far, ESWS submitter plug-ins have been developed for the gLite middleware, the default Metronome job management engine, and for the UNICORE 6 middleware. The UNICORE submitter plug-in uses the UNICORE command line client (UCC) to submit build and test jobs. The UCC offers a set of basic commands as running jobs, getting status, and getting the job output. A JSON[8] job description document is used to specify the executable file, arguments and environment settings, any files to stage in from remote

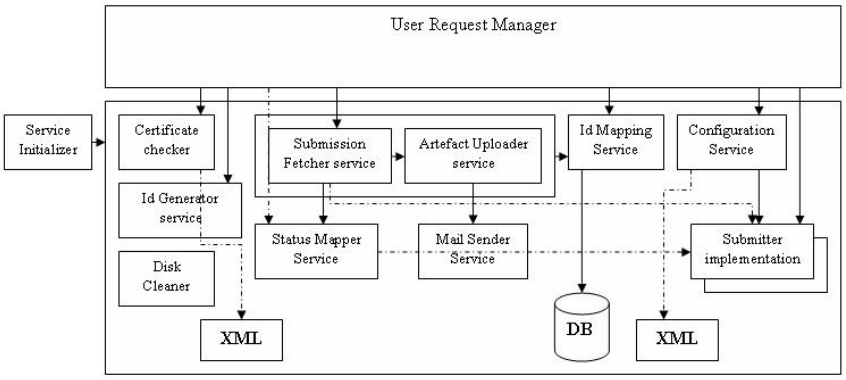

Figure 4. ESWS architecture.

servers or the local machine, and any result files to stage out. Although, jobs may be alternatively specified in the Job standard description language JSDL[9], the UNICORE submitter was designed to use JSON in terms of readability and simpler creation of the job description file.

The UNICORE plug-in has been implemented to perform the following operations when ETICS is executing a build job with UNICORE:

- The plug-in connects to the UNICORE middleware and verifies that the target system is available. The connection is secured by an X.509 based handshake between client and server.
- The JSON file to be submitted to the UNICORE middleware is generated by using the information contained in the ETICS configuration file.
- The actual ETICS job script file to be executed in the job is generated by using the parameters of the submission request.
- The JSON job is submitted asynchronously to the UNICORE 6 middleware by executing the UCC run command. This will download the ETICS command line client, create an ETICS Workspace, get the project of the component to be built or tested, perform the checkout of the component and perform the required build or test. Then the entire workspace folder is added to a tar.gz archive.
- The unique UNICORE ID, which is associated with the submission, is returned to ESWS and is mapped to the ETICS Global job ID.
- The ESWS fetcher service is starting to request UNICORE 6 permanently about job status as long as the execution is running. Then the getting output operation of the UNICORE plug-in is invoked to transfer the results to an ETICS repository.

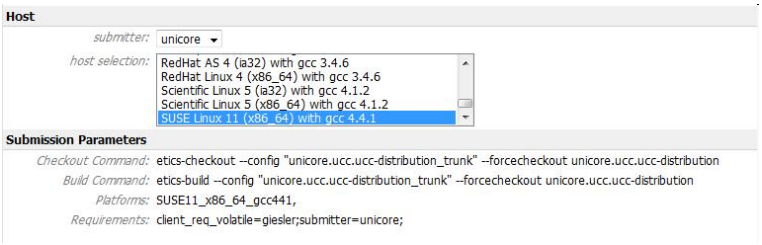

Figure 5. Select ESWS UNICORE submitter for a remote build in ETICS Web client.

Fig. 5 shows again the remote building panel of the ETICS web client known from section 3. This time the user selects the UNICORE submitter plug-in for submitting an ETICS remote build.

Acknowledgments

This work has been funded by the European Union (EU)[11] under contract RI223782. The grant period was from 01.03.2008 until 28.02.2010. We would like to thank CERN staff, especially Andres Abad Rodriguez, for the support throughout. Testbed ressources were provided by CERN, Forschungszentrum Jülich, and INFN-CNAF.

References

1. ETICS Website. `http://etics.web.cern.ch/etics/`
2. UNICORE Website. `http://www.unicore.eu/`
3. EGEE Website. `http://public.eu-egee.org/`
4. AWS Website. `http://aws.amazon.com/de/`
5. Sourceforge Website. `http://sourceforge.net/`
6. ETICS A-QCM feature. `http://etics.web.cern.ch/etics/aqcm.htm`
7. gLite Grid middleware. `http://glite.web.cern.ch/glite/`
8. JSON - (javascript object notation) - a lightweight data-interchange format. `http://www.json.org/`
9. A. Anjomshoaa, F. Brisard, M. Drescher, D. Fellows, A. Ly, S. McGough, D. Pulsipher, and A. Savva. *Gfd-r.056, job submission description language (jsdl) specification* (version 1.0, November 2005).
10. A. Pavlo, P. Couvares, R. Gietzel, A. Karp, I. D. Alderman, Livny M., and C. Bacon., *The nmi build and test laboratory: Continuous integration framework for distributes computing software. In the 20th conference on Large Installation System Administration* (Washington, DC, pages 263–273, December 2006).
11. European Union (EU). `http://ec.europa.eu/`

HiLA 2.0 – Evolutionary Improvements

Björn Hagemeier[1] **and Roger Menday**[2]

[1] Jülich Supercomputing Centre,
Research Centre Jülich, 52425 Jülich, Germany
E-mail: b.hagemeier@fz-juelich.de

[2] Fujitsu Laboratories of Europe
Hayes Park Central, Hayes End Road
Hayes, Middlesex, UB4 8FE
E-mail: r.menday@fle.fujitsu.com

HiLA is a High Level API for writing Grid applications. It has been actively used by a number of communities and projects, for example several EU funded projects such as A-WARE[1] and eDEISA[2]. It has also been used in the German funded D-Grid project AeroGrid[3]. This maturity in the past few years has seen the HiLA software transition to the next major version - version 2.0. We report on this in this paper.

1 Introduction

HiLA is a Java API library which designed to assist writing applications which utilise distributed resources on the Grid. The API is Java based, aiming for a high-level of abstraction offering the programmer concise and easy to read code. This encompasses all of the usual tasks of job, data and information management. Following a clear interface first design strategy, the API is able to support multiple UNICORE Grid environments through multiple implementations. However, as the middleware has advanced to the current version 6 of UNICORE, the quality of the service from the underlying middleware has improved, and in addition several additional API improvements have been made possible. This prompted version 2.0, a significant but evolutionary step.

The HiLA 1.0 branch of developments is quite mature and keeps a lot of the original developments from when HiLA was still called Roctopus[4,5]. Since then it has been used in several projects, and feedback has been incorporated in the new developments. The HiLA 1.1 branch adds support for multi-user environments on top of 1.0. This has successfully been used in the European DEISA[6] and eDEISA projects for the implementation of portals to grant access to Grid resources.

Out of the experiences gained with these versions of HiLA, we started designing HiLA 2.0 with the following goals in mind:

- The use of the API should be further simplified.
- Instantiation of resources should take as few lines of code as possible.
- The API should be able to support multiple users, perhaps through an extension of the Location structure.
- Further emphasis of the generic aspects of the API.
- Ability to interact with the management of long running activities.

Furthermore in this paper, in order to demonstrate the capabilities of the HiLA API, we will demonstrate a Grid Shell application that has been implemented using the API. A vast number of commands similar to a UNIX Shell or remote file system clients such as FTP clients are available and allow the user to use remote systems intuitively. The demonstration shows some exemplary commands of the HiLA Shell to demonstrate how easy it is to implement such functions based on the API. The demonstration is also available as a movie[7].

2 Improvements

Inconveniences in using the HiLA API were noted as it has been put to use. They will be explained in more detail below.

We noticed that the API always had to be instantiated in the same way and there was some redundancy in this instantiation. Also, the Location class was not as extensible as could be. At the same time, configuration had been realized using the Spring framework, causing some overhead when the API was instantiated and posing a barrier for users not acquainted with Spring configuration and XML files. During runtime we noticed problems with misbehaving Grid sites that caused HiLA based clients to be unresponsive. If HiLA was used purely based on the locations, the resolution of these locations into usable Grid objects was very costly. Therefore caching mechanisms have been introduced, such that objects could be resolved faster once discovered for the first time.

2.1 Ease of Use

Considering the following code example:

```
Location loc = new Location("unicore6:/sites");
Locatable locatable =
          HiLAFactory.getInstance().locate(location);
```

Making note of the HiLAFactory.getInstance() factory code, it is repetitive, as it is needed whenever a Grid object is to be resolved given its location. At the same time, it doesn't require any contextual information in its call. Code like this should be hidden somewhere, thus it seemed only natural to move this code into the Location class and have it resolve the actual objects.

The following code demonstrates this:

```
Location loc = new Location("unicore6:/sites");
Resource res = loc.locate();
```

Not only is this code shorter than the original example, but it also involves one less class that the user of the API has to know. The reader will also notice that the Location.locate() method now returns a Resource instead of a Locatable. This has been renamed, because Resource seemed a more intuitive concept than Locatable.

2.2 Genericity and Extensibility

Another change that also involved the Location class is the improvement of its extensibility. Originally, there were a number of methods helping to distinguish among the various types of Grid artifacts. The following listing shows some of them.

```
Location loc = new Location("unicore6:/sites");
boolean isGridLoc = loc.isGridLocation();
boolean isSiteLoc = loc.isSiteLocation();
boolean isFileLoc = loc.isFileLocation();
```

Providing the methods in this manner is restrictive for several reasons. First of all, any newly introduced resource type would break the interface, as it would require a new method along the lines of Location.isXYZLocation(). Thus, the Location class has to be changed to allow for the introduction of new resource types. Even if the structure of the location of a particular type changed, these methods need to be updated.

Our goal was to provide a generic Location class that is agnostic of any type specific information, yet can still provide answers to the simple question whether a given Location is of a particular type. Thus, a new boolean method Location.isLocationOfType(Class) was introduced and replaces the type specific methods mentioned above.

```
boolean isFile = loc.isLocationOfType(File.class);
```

Naturally, the Location class needs additional aid in order to determine the type of the particular Location. This is provided by a ResourceTypeRegistry that is instantiated when the HiLA library is first accessed during the runtime of an application.

Each resource type is annotated with a list of regular expressions matching Locations of this resource type. These regular expressions, when matched, map a Location to the resource type. The following listing displays such an annotation:

```
@ResourceType(locationStructure = {
 "unicore6:/sites/{site}/storages/{storage}/files((/?)|(/.*))",
 "unicore6:/{user}@sites/{site}/storages/{storage}/files((/?)|(/.*))",
 "unicore6:/sites/{site}/tasks/{task}/wd/files((/?)|(/.*))",
 "unicore6:/{user}@sites/{site}/tasks/{task}/wd/files((/?)|(/.*))",
 "unicore6:/storages/{storage}/files((/?)|(/.*))",
 "unicore6:/{user}@storages/{storage}/files((/?)|(/.*))" })
public class Unicore6File extends BaseFile implements File {
...
```

As you can see, there is more than one regular expression matching the location structure of a UNICORE 6 File. Care must be taken that the regular expressions for the location structure of different types of resources are mutually exclusive.

To be precise, the Strings inside locationStructure are not regular expressions themselves, but they will be converted into regular expressions by HiLA internally. The parts within curly braces, e. g. {storage}, are variables, which will be filled with the values of a concrete Location String in the position of the variable.

Figure 1 shows how the new module structure supports extensibility. The base module "hila-api" provides basic functionality to manage Resources and is agnostic of any grid specific information. Based on it is the "hila-grid-api" module, which provides Grid specific

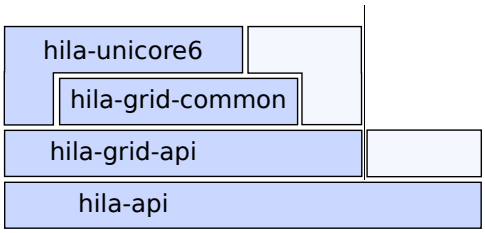

Figure 1. Separation of modules in HiLA. Pale areas represent extension points, i. e. there could be other implementations of the Grid API or entirely different abstract APIs based on the resource concept provided by hila-api.

interfaces that already existed in HiLA 1.x. "hila-grid-common" provides basic implementations of these interfaces, which lead to homogeneous default behavior across different middleware specific implementations, one of which is "hila-unicore6". Implementations for other Grid middleware are planned and even entirely different abstract APIs dealing with different resource types can be envisioned.

2.3 Configuration

HiLA 1.0/1.1 expected a Spring[8] configuration for any of the implementations. This was detrimental for several reasons. For one, Spring introduced a significant overhead in startup time, making HiLA less usable when used in command line utilities that would have to load the full configuration on each startup. Additionally, when HiLA was used in the eDEISA portal[9], the Spring configuration for HiLA itself and the Spring configuration for the portal container were in conflict with one another. This would also be true for any other such setting.

Therefore, it was decided to resort to a properties base configuration using simple pairs of names and values. The following is an example of such a configuration for UNICORE 6.

```
hila.unicore6.registries = \
    https://localhost:8080/DEMO-SITE/services/Registry? \
    res=default_registry

hila.unicore6.keystore = /home/user/.hila2/demo.jks
hila.unicore6.alias = demo user
hila.unicore6.password = the!user
hila.unicore6.keystoretype = jks
hila.unicore6.truststore = /home/user/.hila2/demo.jks
hila.unicore6.truststorepassword = the!user
hila.unicore6.truststoretype = jks
```

2.4 Optimizations for large infrastructures

Outside of any well-behaved environment with no network problems or non-responding sites, it became clear that in large infrastructures the discovery of sites can pose problems.

First of all, network communication with each of the sites is necessary, at least in order to find out whether it is available and responsive. Doing this in a sequential manner would take long and can be improved by connecting to the sites in parallel. This approach does not only reduce the time to access sites, it can also deal with misbehaving sites that cause connection time outs.

Suppose each site would only respond slowly to any request, and these requests were done sequentially. The long waiting times to access each of the sites would easily add up to several minutes and thus render any client based on HiLA unusable. The likelyhood of the existance of misbehaving or slow sites in large environments grows, so this is another reason why concurrent site discovery is needed.

Secondly, even if sites fail to respond completely after a long timeout, this does not stop the other sites from being used even before the long timeout of that one site.

Doing the discovery of sites in parallel also allows us to set a general limit for the timeout of site discovery irrespective of the number of sites or individual network timeouts.

2.5 Caching

Even if resources can be located rather quickly, as described in the previous section, it is still desirable to cache local representations of remote Grid resources. Resource discovery over network always takes time, which can be avoided by caching the information about it. This is particularly true, if resources are to be discovered based on their HiLA location.

It is assumed that resources of the same type follow the same caching strategy, e. g. sites are more durable resources than files in a storage or tasks that may come and go. Therefore, the implementation class of the resource type to be cached is responsible for caching.

3 Conclusions and Future Work

In addition to the improvements described above, the Grid interfaces have been extended in order to support more fine grained control of what happens in the background, i. e. explicit invocation of recursive operations on directories, explicit overwrite of already existing files etc.

The original HiLA API managed data transfers based on files, which is inappropriate if HiLA is used in places where the data should be managed in 'pass-through' manner, e. g. in a portal. Therefore, we will offer data transfers based on streams, such that this becomes possible.

Lastly, the API has been re-engineered to use the java . util . concurrent interfaces. This is beneficial whenever one needs to run numerous tasks concurrently and handle their results. This pattern can also be applied for the execution of remote Grid tasks. An experimental implementation of the MapReduce method already used this technique.

Different implementations of the Grid API may not support all of the abstract features. An application based on HiLA should be able to find out which features are supported by a certain implementation. The notion of capabilities supported by an implementation will be investigated. This feature has not been realized yet. Until then, we confined ourselves to throwing a 'not supported' exception, should an action not be possible in an implementation.

The caching mechanism, while effective, is currently implemented in a naive manner. The memory consumption increases with the number of locations known to the system. This approach does not scale and we intend to introduce a caching library like EHCACHE[10] to deal with this. Using it, persistence of discovered resources beyond the runtime of a HiLA application will be easy to implement as an additional feature.

Since its inception, HiLA has evolved and has been responsive to user requests. A second major version of the software has been released. The power of the resource-oriented model has been emphasized, the API has been simplified further and significant performance improvements have been realised.

References

1. A-WARE project site – `http://www.a-ware-project.eu/`
2. eDEISA public deliverables – `http://www.deisa.eu/publications/deisa1-public-deliverables#edeisa`
3. AeroGrid project site – `http://www.aero-grid.de/`
4. Menday, R.; Kirtchakova, L.; Schuller, B. & Streit, A.
   "An API for Building New Clients for UNICORE"
   Bubak, M.; Turala, M. & Wiatr, K. (ed.)
   Proceedings of the Cracow Grid Workshop '05, Academic Computer Centre CYFRONET AGH, Cracow, Poland, **2006**, pages 240-245.
5. Hagemeier, B.; Menday, R.; Schuller, B. & Streit, A.
   "A Universal API for Grids"
   Bubak, M.; Turala, M. & Wiatr, K. (ed.)
   Cracow Grid Workshop '06, Academic Computer Centre CYFRONET AGH, **2007**, pages 312-319.
6. Distributed European Infrastructure for Supercomputing Applications – `http://www.deisa.eu`
7. HiLA Shell demonstration movie – `http://www.youtube.com/watch?v=STlWveNd4m0`
8. SpringSource.org – `http://www.springsource.org/`
9. The DEISA Life Sciences Portal Architecture – `http://www.deisa.eu/publications/deliverables_fp6/eDEISA/eDEISA-eSA3-A1.pdf`
10. EHCACHE project site – `http://ehcache.org/`

MMF: A Flexible Framework for Metadata Management in UNICORE

Waqas Noor[1] **and Bernd Schuller**[2]

[1] Rheinisch-Westfälische Technische Hochschule (RWTH),
D-52056 Aachen, Germany
E-mail: {waqas.noor}@rwth-aachen.de

[2] Jülich Supercomputing Centre,
Research Centre Jülich, 52425 Jülich, Germany
E-mail: {b.schuller}@fz-juelich.de

UNICORE is a state of the art, ready-to-run Grid middleware, which provides services to manage distributed resources and applications seamlessly and securely with utmost ease. The UNICORE storage service manages data but currently searching for data is limited to basic metadata only. Metadata has paramount importance in discovering data and providing powerful searching capabilities on operating systems, Grids and the World Wide Web (WWW) so there is an explicit need for metadata management in UNICORE.
In this paper, we propose a new approach to manage metadata in UNICORE, which keeps the metadata close to the data. As a realization, we present a Metadata Management Framework (hereafter referred to as MMF) - a flexible framework for metadata management. Extending the UNICORE storage service with the MMF, we are able to support searches on data using basic metadata as well as full-text search. The MMF allows to tag data with a user defined vocabulary. MMF is flexible and extensible since it does not require any schema to manage the metadata. We present different usage scenarios of MMF and elaborate metadata management within UNICORE's storage service. We demonstrate scalability and flexibility of our framework by examining different scenarios. Finally, we compare MMF with related technologies.

1 Introduction

Rapid technological advancements and active grid community enable grid infrastructures to solve huge computational challenges. Solving these challenges requires a lot of computational power and huge storage which cannot be provided with a single machine. Today, Grids do not provide only computational services but also data management services. Service oriented architectures provide new ways to utilize grids as storage systems and computational machines as separate services. Thus, new requirements for managing the data and metadata in the context of grids emerged. Current state of the art solutions of metadata management treat the data and metadata as two different entities and store them separately. Metadata describes data and both are tightly coupled.

As we stated earlier that the metadata of resources are tightly linked with each other and managing data and metadata together lead towards a new set of challenges. These new set of requirements open the areas of research and hence require to re-evaluate the existing metadata management solutions in Grids. Presently, the metadata management systems manage the metadata separately from the contents of data or resources[a].

[a]The word *Resource* is used as an entity which could be anything and identified by unique key. In our case it is mainly a file.

We believe that metadata of the resources should be managed along with the resource or at least should stored close to the resource on storage systems. Our hypothesis to manage the metadata together with the resources is backed by the design and implementation of new framework presented in the rest of the paper.

The paper is organized as follows: In the next section, we discuss the related work. In Section 3, we presented the UNICORE architecture and elaborated its Storage Management Service. In Section 4, MMF's architecture is elaborated with reference to UNICORE Storage Service. This is followed by usage scenarios of MMF in Section 5. We discuss the evaluation of our framework in Section 6. Finally, Section 7 concludes the paper and also points to future work.

2 Related Work

Several systems and implementations exist in the area of metadata and grids. These works mainly follow the centralized approach to manage the metadata. The data and metadata is de-coupled and stored in different repositories[b].

Metadata Catalog Service (MCS) is a metadata management standalone application which is also integrated in OGSA-DAI[1]. The MCS work is based on a schema to store the pre-defined metadata into categories such as Logical File Metadata, Logical Collection Metadata, Logical View Metadata, etc. The user-defined metadata attributes, which are important for the support of application-specific domain, are also stored into standard schema[1]. MCS stores the metadata in a database. The web server and client API provide interfaces to access the metadata.

The Chemomentum data services are UNICORE-based flexible solution for managing large amounts of data and metadata produced in a Grid[2]. The metadata in Chemomentum project is maintained in a centralized fashion, i.e. rational databases with pre-defined attributes.

The Integrated Rule-Oriented Data System (iRODS), is a data grid software system developed by the Data Intensive Cyber Environments research group (developers of the SRB, the Storage Resource Broker) and collaborators[3]. The iRODS mainly tends to solve the distributed data management system constraints rewritten problems[4]. The iRODS provides the metadata management such as System Metadata (user, file and storage name space) and Domain Metadata (key/value pairs, domain specific). The iRODS metadata functionality keeps the records of replication of data, state changes, etc. It manages the metadata in centralized manner.

3 UNICORE Architecture

UNICORE (**UN**iform **I**nterface to **CO**mputing **RE**sources) is ready-to-run open source grid middleware which provides seamless, secure, extensible and intuitive access to grid resources. The aim of UNICORE is to access not only supercomputers, clusters but personal computers, transparently and elegantly. By the time of writing, the latest version, UNICORE 6 services conforms to the specification of the Open Grid Services Architecture (OGSA) [5] [9]. UNICORE middleware is composed of different components, which can

[b] Data is stored at file systems and metadata is stored in database systems.

be divided into 3 layers due to their nature/services of these components. *Client layer* provides the necessary tools to end users of grids to access the resources. The *service layer* hides the business complexity from the clients. The *system layer* facilitates the service layer by giving access to the resources. The details of each layer and components are given in the subsequent paragraphs.

UNICORE's *client layer* has various types of clients, targeting users of experienced grid knowledge to novice [6]. UNICORE Rich Client (URC) is a eclipse based graphical client which hides all the complexity from users i.e. creation the job, submission, etc. It provides the simple interfaces to access the grid resources. UNICORE CommandLine Client (UCC) provides the basic interface for managing the jobs on grids. UNICORE services can be accessed through High Level API for Grid applications (HILA) and web portal [8].

UNICORE's *service layer* consists of services and components which mostly conform to OGSA specifications and Web services are based on Web Service Resource Framework (WSRF) [8]. Before UNICORE 6, previous versions had some proprietary components called UNICORE Atomic Services (UAS). Some of these proprietary components i.e. UNICORE Storage Management Service (SMS), File Transfer Service (FTS) still exist in the latest UNICORE. The UNICORE services are secured through well known X.509 certificates. All the requests to access the UNICORE' services have to be passed through Gateway which ensures the authentication of users.

At the bottom layer of UNICORE architecture, *system layer*, TSI (Target System Interface) component is interacting with the UNICORE services and physical resources on grid, operating system, etc. At this point, the jobs are executed as per their job definition. For each job, a dedicated directory is created for staging the data *IN* and *OUT* (moves the data into the working directory and fetches from working directory respectively).

To sum up, the client creates a job and then Gateway authenticates the request, later it delegates the request to *unicorex* server. Further, it uses the IDB (Incarnation Database) for mapping the user request for Target System Interface (TSI). The TSI from system layer takes the request from service layer component for final execution. A separate USpace [7] is created for each job as a working directory.

4 MMF Architecture

The architecture of Metadata Management Framework (MMF) is structured as two layered architecture, defined as *Client Layer* and *Service Layer*. These layers are modeled as Service Oriented Architecture which also conforms to the OGSA standards. Figure 1 illustrates the different component of the framework.

As shown in Figure 1, clients and Metadata Services are loosely coupled and modeled as Service Oriented Architecture. The client plays role as a service consumer where as Metadata Service act as service provider.

Client Layer consists of SWT (The Standard Widget Toolkit) based graphical client which consumes the service. The client wraps the complexity of accessing the Web service and provides the interfaces to work with the Metadata Service. In addition to CRUD operations of Metadata Service, it also provides two search interfaces for searching the resources i.e. *simple* and *advance* search respectively.

Service Layer consists of implementation of *Metadata Service* along with *Metadata*

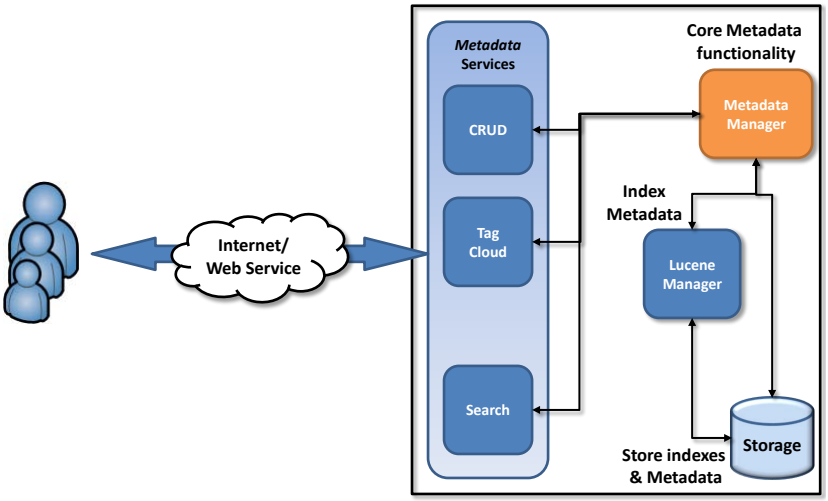

Figure 1. Metadata Management Framework Architecture

Manager and *Lucene Manager*. The *Metadata Service* is an interface between the core functionalities and clients. It exposes the core functionalities as web services. Each service is uniquely identified in service WSDL (Web Services Description Language) and is accesses via the Internet or Intranet through SOAP (Simple Object Access Protocol) messages. The *Metadata Manager* component provides the implementation of Metadata Service interface. It performs the core functionalities of metadata management service and also communicates with the Lucene Manager. *Lucene Manager* component is used for indexing and retrieval of metadata from storage. The Lucene Manager relies on the third party open source indexing technology called, Apache Lucene[c].

4.1 Metadata service in UNICORE Storage Service

The Metadata Service is integrated with UNICORE Storage Service. The extended Storage service (storage with metadata) is supported by metadata operations. The usage scenarios of Metadata Service are explained in detail in Section 6.1. Instead of Metadata graphical client, one of the UNICORE client invokes the Storage Service shown as step 1 in Figure 2.

UNICORE Storage service creates metadata objects for the request in step 2 as shown in Figure 2 and delegates the request to Metadata Manager shown in step 3 in Figure 2. The Metadata Manager is responsible for performing the core metadata operations i.e. *Create*, *Update*, *Search*, etc. The metadata is indexed as shown in step 4. Finally the index is store on storage in step 5.

JSON (JavaScript Object Notation) is used as a data structure to manage the metadata as well as manipulating the metadata objects. Every metadata object is represented as

[c]Apache Lucene is a high performance open source api for indexing, searching and retrieval of documents `http://lucene.apache.org/`.

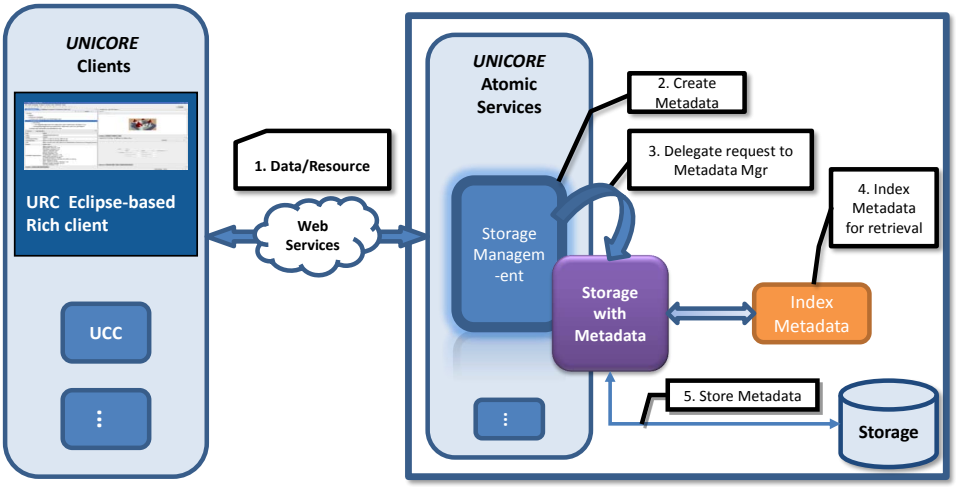

Figure 2. Metadata Service within UNICORE' Storage Service

key/value pair in JSON. A special key, ***tag***, is used to manage the user defined metadata. The sample JSON representation of metadata objects is shown below:

Listing 1. JSON representation of an object.

```
{
 "resourceName": "/./.TimeSeriesData",
 "owner":"username>",
 "experimentPlace":"FZJ",
 "desc":"Multiple dipole construction method using MUSIC",
 "tags":"Time series, parallel approach, estimation, noise"
}
```

5 Evaluation

In the subsequent sections, different performance aspects of Metadata Management Service are elaborated. We tested the system at features level (in comparison with UNICORE' SMS) as well as at use cases level. The functionality of Metadata Service is loosely bound to indexing technology (provided through Apache Lucene), so we also test the performance of underlying technology as a part of the metadata service. The performance measurements are reported in the following sections.

5.1 Comparison with UNICORE's SMS search feature

UNICORE Storage Service provides the searching interface on top of storage system. The *extended* Storage Management Service has various types of search capabilities. The

Search by	Storage Management Service (SMS)	*Extended* Storage Management Service
Basic File Metadata (filename, creation date etc)	✓	✓
Boolean Query	Partially (AND, OR)	✓
Wild Characters search on text	Partially (filename only)	✓
Range Query	Partially (date only)	✓
Fuzzy Query	X	✓
Full Text Search	X	✓

Figure 3. Search in UNICORE Storage Management System in comparison with the Extended Storage System

overview of search capabilities in comparison with extended Storage Management Service is shown in Figure 3. At present, UNICORE version (6.2.2) does not provide text based search in its storage service. With our approach, the *extended* Storage Management Service provides the text based search as well. For example, the searching interface is capable of searching the content by providing the boolean query. In contrast to *boolean query* in SMS, our solution allows boolean operator, *NOT* in the queries to exclude the search results. Similarly, the resources can be searched by their contents with *wild characters* queries. The current implementation of UNICORE Storage Service does not provide the search on files by fuzzy search as shown in Figure 3. For example, the resources which contains "*DNA*" can be searched by fuzzy query "*BNA*".

Before delving into use case level performance benchmarks, we state the testing environment: The total number of documents are 60K. The documents type is text and total size is about 27.5 GB. The index size is about 1.1 GB and we only indexed 2 fields. Their contents are not stored at index. The StandardAnalyzer from Apache Lucene is used for tokenizing the documents.

Figure 4 shows the results of query against the different sizes of indexes. As first entry shows that creating the instance of IndexSearcher class takes *132* milliseconds. The instance of QueryParser class takes *26* milliseconds. The parsing of query and performing search takes 15 and 9 milliseconds respectively. The IndexSearcher takes most of time to perform the search. The instances of IndexSearcher class as well as QueryParser class can be created once and used in search methods.

The instance of *ParseQuery* cannot be created once because it translates the string representation of query to Lucene ParseQuery instance. In some cases, where query remains unchanged, the instance of ParseQuery can be created once as well. If we exclude the time taken by the IndexSearcher and QueryParser, it takes *24* milliseconds to find the word in the index of *10K* documents as shown in Figure 4 (The time taken by ParseQuery and Search). The results are quite similar to other cases except the time taken by the IndexSearcher instance increases gradually starting from *132* milliseconds to *195* milliseconds in our test.

6 Usage scenarios

In this section, we will elaborate the usage scenario of Metadata Service as a loosely coupled service or as an integrated service with UNICORE Storage Management Service. In

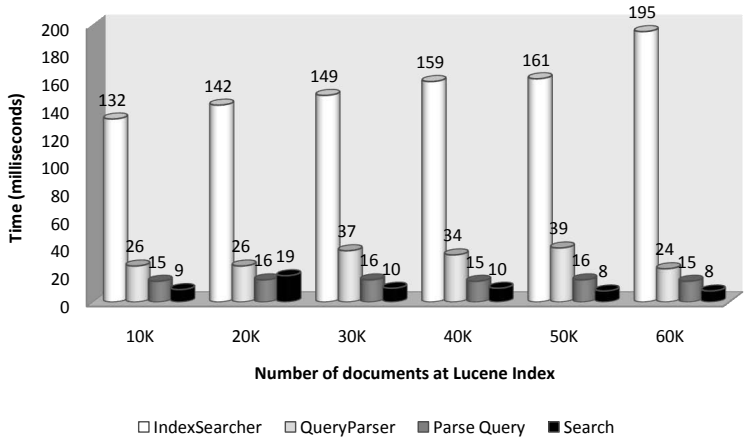

Figure 4. Search the text/contents by any word.

subsequent sections, we will elaborate both cases with simple scenarios.

6.1 UNICORE Storage Service with metadata

The Metadata Management Framework integrated in UNICORE Storage Service. Such usage of metadata management as a module within UNICORE Storage Management Service is shown in Figure 5. In this scenario, illustrated in Figure 5, Metadata management capa-

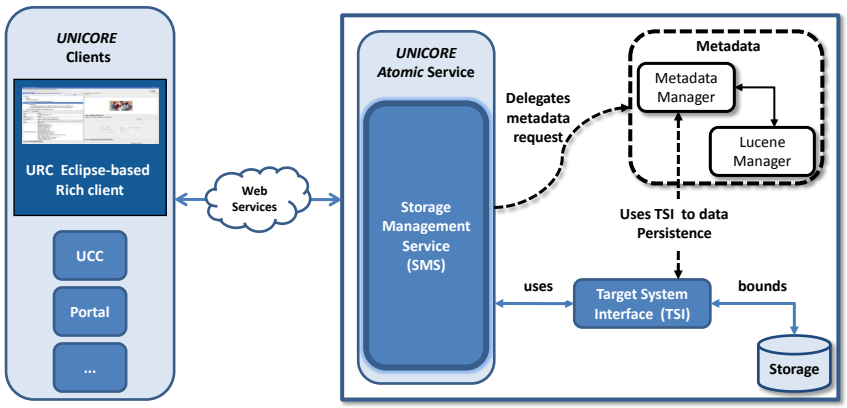

Figure 5. Metadata service integrated in SMS

bilities are integrated within the Storage Management Service. The details and sequence of actions in such usage scenario are given below:

1. UNICORE clients consume the UNICORE Services which are exposed as web services.
2. In the current implementation of our work, we extended the *UNICORE Storage Management Service* for metadata.
3. Storage Service delegates the file operations request i.e. import/export data, search and delete files, etc to Metadata Manager.
4. Metadata Manager performs its tasks (CRUD operations and search).
5. Metadata Manager uses the *Target System Interface (TSI)* storage for storing and retrieving the Metadata from storage.
6. Storage Management also uses the TSI storage for data management.

6.2 Metadata Catalog Service

Metadata Management service is based on Service Oriented Architecture (SOA). The metadata service can be seen as a loosely coupled service from UNICORE Storage Management Service. In this scenario which is presented in Figure 6, the metadata service can be deployed and consumed without the UNICORE Storage Service. Currently, UNICORE storage management service is consumed for storing the catalog. We see in Figure 6 that

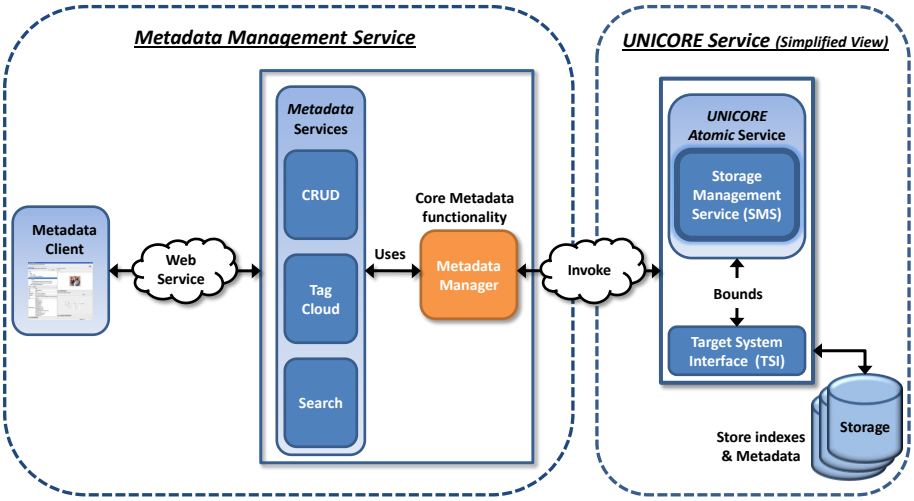

Figure 6. Metadata Catalog service using UNICORE Storage Service

UNICORE Atomic Service shown as a separate part which is loosely coupled with Metadata Services through Web services. The complete scenario is given below:

1. Metadata Client consumes the *Metadata Services*.

2. Metadata Services are backed by the Metadata Manager which provides the core functionality to create, update, delete and search metadata.

3. For storing and retrieval of metadata, *Metadata Manager* uses the UNICORE Storage Management.

4. Storage Management Service is responsible for storing and retrieving the indexing and metadata from the storage system.

7 Conclusions and Future Work

We presented a metadata management solution for a state of the art grid middleware, UNICORE. The metadata management solution can be used in two ways: integrated with the UNICORE Storage Service or as an independent Metadata Catalog Service. The paper work presented a novel way to manage metadata in grids, and a catalog service which can serve as a contribution to other research activities in the grid community.

In this paper we also presented the metadata service as a Catalog Service. It means that research community is not required to use the UNICORE Storage Management Service for managing the metadata. The metadata service can be accessed independently from the UNICORE's services, realizing as a Catalog Service.We compared the features of our system with those of the UNICORE Storage Management Service and related technologies and highlighted the suitability and differences. The results of the testing have been presented in the Evaluation section.

The results from the evaluation of the system, we conclude that the design is quite flexible which allows easy integration with with existing technologies and other middlewares. Our solution is dependent of the underlying technology which indexes the data and metadata. Based on the results from the underlying technology in our testing environment we show the suitability of the solution for many applications, ranging from small standalone applications to enterprise solutions to the grid applications.

Currently, the metadata service only indexes text documents which are in English language. For certain applications, it may be required to deal with other languages. In future work, we intend to provide indexing options for multiple languages.

Furthermore, the evaluation results show that metadata service spends largest amount of time in indexing the huge documents which may leads to drawback for time critical applications. The indexing time can be improved by several optimization techniques provided by the underlying technology. Nevertheless, the indexing process should be executed asynchronously for time critical applications in future work.

References

1. G. Singh, S. Bharathi, A. Chervenak, E. Deelman, C. Kesselman, M. Manohar and S. Patil, L. Pearlman, *A Metadata Catalog Service for Data Intensive Applications*, SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing.
2. K. Rasch, R. Schöne, V. Ostropytskyy, H. Mix, M. Romberg, *A Metadata Catalog Service for Data Intensive Applications*, Euro-Par Workshops: The Chemomentum Data Services - A Flexible Solution for Data Handling in UNICORE, Proceedings

of 4th UNICORE Summit 2008, Euro-Par 2008 Workshop proceedings, pp.84-93, Springer LNCS 5415, 2008.
3. M. Hedges, T. Blanke, A. Hasan, *Rule-based curation and preservation of data: A data grid approach using iRODS*, Future Generation Computer Systems **446-452**, 2009, .
4. A. Rajasekar, M. Wan, R. Moore, W. Schroeder, *A prototype rule-based distributed data management system*, HPDC workshop on Next Generation Distributed Data Management, Paris, France, 2006.
5. I. Foster, C. Kesselman, J. Nick, S. Tuecke, *The Open Grid Services Architecture V.1.5*, Open Grid Forum Recommendation GFD.80, http://www.ogf.org/documents/GFD.80.pdf, 2006.
6. A. Streit, *UNICORE: Getting to the heart of Grid technologies*, eStrategies report, `http://www.unicore.eu/documentation/files/streit-2009-EP.pdf`, 2009.
7. M. Romberg, *The UNICORE Grid Infrastructure*, Special Issue on Grid Computing of Scientific Programming Journal, 10, pp. 149 -157, 2002.
8. W. Bari, A. S. Memon, B. Schuller, *Enhancing UNICORE Storage Management using Hadoop Distributed File System*, Euro-Par 2009, Parallel Processing - Workshops: HPPC, HeteroPar, PROPER, ROIA, UNICORE and VHPC, Revised Selected Papers, Lecture Notes in Computer Science, LNCS 6043, Springer, 2010.
9. Ian Foster, Carl Kesselman, Jeffrey M. Nick, Steven Tuecke, *The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration*, Global Grid Forum, 2002.

Extended Execution Support for Scientific Applications in Grid Environments

Bastian Demuth, Sonja Holl, and Bernd Schuller

Jülich Supercomputing Centre,
Research Centre Jülich, 52425 Jülich, Germany
E-mail: {b.demuth, s.holl, b.schuller}@fz-juelich.de

In high performance computing (HPC), there is traditionally a very high heterogeneity of hardware and software specialising on different application scenarios. Additionally, there are various ways of accessing HPC resources and scheduling computational jobs via different batch systems. Grid middleware systems like UNICORE[1] try to overcome these obstacles by providing unified high-level interfaces that can be used without any knowledge of the specific job scheduler in place. UNICORE 6 also introduces *applications* as abstractions for executables, so users do not have to know the absolute path to an executable (which can vary with the system). The work described in this paper introduces the concept of configurable execution environments to the UNICORE 6 middleware. Execution environments can be used to run applications in different modes. The flexibility gained through this mechanism helps reduce the need for user-defined application wrapper scripts and adds a new level of user-friendliness.

1 Introduction

UNICORE 6 is able to map applications to concrete executables by evaluating its *incarnation database* (IDB) before forwarding incoming jobs to the underlying batch system. The contents of the IDB are configured via an XML[2] file. This allows system administrators to define which applications are installed at their site and how these applications should be invoked. Such information is used to hide the local peculiarities of a site thus leading to more platform independent job descriptions. However, the original IDB implementation does not provide a standardized way to run applications in different execution modes (e.g. *debug*, *timed*, *parallel*). An execution mode can be more than a simple flag, it might itself be configurable with different parameters and options. For example, the *parallel* execution mode requires the number of parallel tasks as a parameter. Instead of being run in a specific mode, a main application could also require additional software tools to be loaded or specific environment variables to be set before it is started. Trying to cover these different scenarios with a single flexible mechanism, we developed the notion of an *execution environment*, which is slightly more abstract than the term 'execution mode'. In order to enable system administrators to define execution environments that can be selected and parameterized by end-users, both the server and client side of the UNICORE 6 middleware were extended. Two main use cases were defined to drive the development: the support for a parallel execution mode based on the popular Message Passing Interface (MPI[3]) and dynamic loading of software modules in the DEISA[4] infrastructure. Section 2 describes these use cases in detail. Section 3 lists the main concepts behind execution environments in UNICORE 6 and explains how they should be defined by administrators. Section 4 sketches the extensions to the UNICORE 6 server, whereas Section 5 focuses on additions to the client side. Section 6 summarizes the results.

2 Motivation and Use Cases: MPI and DEISA Modules

In high performance computing, many applications use the Message Passing Interface for the intercommunication of parallel tasks on supercomputers. There are several MPI implementations, e.g. OpenMPI[5] or MPICH2[6] which work slightly different and have to be configured in slightly different ways. Before the advent of the new execution environment extension, users had to know about the availability and configuration details of MPI implementations on target systems. Wrapper scripts had to be written for choosing the desired MPI implementation and setting its parameters (e.g. the number of parallel processes).

Executing shell scripts via UNICORE 6, instead of using UNICORE's application abstraction to invoke the desired executable directly, is possible but yields a number of drawbacks: First, the full executable path is normally used inside the script, which makes the script more system dependent. Second, UNICORE's ability to provide graphical user interfaces (GUIs) tailored to different applications is not utilised, the user will just see a simple GUI for entering a shell script. Third, programming experience in a scripting language is necessary for writing wrapper scripts. Finally, this method requires knowledge of the specific MPI implementation at the target system and its parameters. In result, the end-users' benefit gained through UNICORE was quite small for MPI-based applications.

One way of solving this problem would have been to let administrators provide the required wrapper scripts on the server side. This solution, however, easily leads to code duplication (similar code for configuring the MPI implementation is spread across various wrapper scripts). Defining a configurable execution environment that can be used for running an application with MPI is a much cleaner approach: The execution environment is defined only once and can be freely combined with any application.

As stated above, the second use case that drove the development was dynamic loading of DEISA software stack modules providing a wide range of tools that can be utilised by scientific applications. The DEISA software stack includes modules for compilation (Java, C/C++, Fortran, etc.) mathematical operations (e.g. Fast Fourier Transform, linear algebra), profiling and performance measurements, and other purposes. Before a main application can use one of these software modules, the module has to be loaded explicitly, which usually happens through a shell command. Again, this lead many users to wrapping their main application call inside a shell script and preceding it with the necessary commands to load the required modules.

With execution environments, loading software modules can be achieved in a much nicer fashion, by modelling each module as a Boolean option defined for each execution environment in the DEISA infrastructure. This way of dealing with module loading has the following advantages: The user gets a graphical overview of all modules that can be loaded plus a short description of each module that can be composed by the administrator. A single mouse click on a graphical checkbox is all the user has to do to make sure a module gets loaded. Most importantly, tailored application specific user interfaces can be employed instead of having the user write a shell script.

3 Defining Execution Environments inside the Incarnation Database

In order to allow administrators to describe execution environments in the UNICORE 6 incarnation database (IDB) file, the schema of this XML file was extended. Execution environment definitions can yield the following information:

- Name: A human readable name for identifying the execution environment, e.g. *debug* or *parallel*.
- Description: A piece of text describing what the execution environment does.
- Executable name: Full path to an executable that implements the logic of the execution environment.
- Arguments: List of command line arguments for the executable. Each argument has a name and an incarnated value, the latter telling UNICORE 6 how to pass the argument to the executable. In addition, arguments may be annotated with metadata:
  - Type: Type of the argument value, can be any of the following: *int* (integer number), *double* (a real number with double precision), *string* (textual value), or *choice* (nominal choice between fixed alternatives).
  - Description: A piece of text that is displayed to the end-user for explaining what an argument does.
  - Valid value range: A character string that allows for validating user input for the argument. For numerical arguments, intervals can be given (using either integer or real values), and nominal values can be defined by a comma-separated list. In the current implementation, string values are not validated, but in the future, regular expressions will be used for this task.
- Options: Somewhat similar to arguments but options can only be enabled or disabled instead of taking a value. Options have metadata, too, but they only contain a description (since options always have Boolean type).
- Precommands: List of shell commands that should be run before the actual executable of the environment; these commands can be enabled and disabled by the user, so precommands and options are very much alike (though the incarnation is different).
- Postcommands: Analogous to precommands but should be executed to perform post-processing after the main application run.

The following example shows the definition of an execution environment for executing parallel applications via MPI (using the OpenMPI implementation). The definition complies to the applicable XML syntax. By inserting it into the IDB file of a recent UNICORE 6 server (execution environment support was added in version 6.3.0) the execution environment is sufficiently described and becomes functional immediately.

```
<jsdl-u:ExecutionEnvironment>
  <jsdl-u:Name>Parallel Execution</jsdl-u:Name>
    <jsdl-u:Description>Run an OpenMPI application</jsdl-u:Description>
    <jsdl-u:ExecutableName>/vsgc/software/openmpi/bin/mpiexec</jsdl-
        u:ExecutableName>
    <jsdl-u:Argument>
      <jsdl-u:Name>Number of Processes</jsdl-u:Name>
      <jsdl-u:IncarnatedValue>-np </jsdl-u:IncarnatedValue>
      <jsdl-u:ArgumentMetadata>
        <jsdl-u:Description>The number of processes</jsdl-u:Description>
        <jsdl-u:Type>int</jsdl-u:Type>
        <jsdl-u:ValidValue>[1,20]</jsdl-u:ValidValue>
      </jsdl-u:ArgumentMetadata>
    </jsdl-u:Argument>
    <jsdl-u:Argument>
      <jsdl-u:Name>Export Environment Variable</jsdl-u:Name>
      <jsdl-u:IncarnatedValue>-x </jsdl-u:IncarnatedValue>
      <jsdl-u:ArgumentMetadata>
        <jsdl-u:Description>Export an environment variable (e.g., "foo=bar
            " exports the environment variable name "foo" and sets its
            value to "bar" in the started processes)
        </jsdl-u:Description>
        <jsdl-u:Type>string</jsdl-u:Type>
      </jsdl-u:ArgumentMetadata>
    </jsdl-u:Argument>
    <jsdl-u:Option>
      <jsdl-u:Name>Verbose</jsdl-u:Name>
      <jsdl-u:IncarnatedValue>-v</jsdl-u:IncarnatedValue>
      <jsdl-u:OptionMetadata>
            <jsdl-u:Description>Be verbose</jsdl-u:Description>
      </jsdl-u:OptionMetadata>
    </jsdl-u:Option>
    <jsdl-u:PreCommand>
      <jsdl-u:Name>Print time before execution</jsdl-u:Name>
      <jsdl-u:IncarnatedValue>date</jsdl-u:IncarnatedValue>
      <jsdl-u:OptionMetadata>
            <jsdl-u:Description>Prints the current time to standard out
                right before execution starts.</jsdl-u:Description>
      </jsdl-u:OptionMetadata>
    </jsdl-u:PreCommand>
    <jsdl-u:PostCommand>
      <jsdl-u:Name>Print time after execution</jsdl-u:Name>
      <jsdl-u:IncarnatedValue>date</jsdl-u:IncarnatedValue>
      <jsdl-u:OptionMetadata>
            <jsdl-u:Description>Prints the current time to standard out
                right after execution ends.</jsdl-u:Description>
      </jsdl-u:OptionMetadata>
    </jsdl-u:PostCommand>
  </jsdl-u:ExecutionEnvironment>
```

Listing 2. Example XML execution environment definition.

In this example all the main concepts behind the definition of execution environments are demonstrated (name, description, executable name, arguments, options, pre- and post-commands).

The following list discusses the XML elements used in the example and explains their effects:

- The *ExecutableName* element points to the *mpiexec* command which is used to initialize a parallel job through the underlying batch system (TORQUE[7] has been installed at this UNICORE 6 site). After calling this command, multiple processes are spawned on allocated cluster nodes, each of which executing the main application. These processes can use MPI for inter-process communication.
- Two arguments for the *mpiexec* command are defined:
  - The *Number of Processes* argument is used to specify how many parallel processes should be spawned. The *mpiexec* command expects this argument to be given in the form '-np numproc' where *numproc* is an integer number. This can be deduced from the text content of the *IncarnatedValue* element. The *Number of Processes* argument is further described by an *ArgumentMetadata* element which provides a text description for the semantics of the argument, declares the argument type (*int* stands for 'integer number') and sets a valid value range for this argument (*[1,20]*). These metadata are later used by the UNICORE Rich Client for building an appropriate user interface and validating user input.
  - The second argument takes character string values and allows for setting additional environment variables for the parallel processes.
- The *Option* element defines an option named *Verbose* that puts OpenMPI in verbose execution mode if selected.
- The precommand defines a shell command that is run on the login node before *mpiexec* is started; analogously, the postcommand is run after *mpiexec* has finished. In our example, these commands simply print the current time to standard out.

4 Extensions to the UNICORE 6 Server

The following additions had to be made to the UNICORE 6 server logic in order to handle such execution environment definitions appropriately:

1. IDB file parsing: The parsing logic for the incarnation database file had to be extended for recognising *ExecutionEnvironment* elements.
2. Information publishing: The UNICORE 6 server publishes all read-in execution environments to the client via a new WSRF[8] resource property.
3. Job description parsing: When the client wants to run a job with a specific execution environment, it adds a small XML snippet to the job description containing the name of the selected environment as well as values for its arguments. The UNICORE 6 server had to be extended to understand these XML snippets correctly and store the contained information for later use during job incarnation.
4. Job incarnation: This term denominates the process of transforming a job description (in job submission description language, JSDL[9]) into a fully qualified executable shell script that is handed over to the batch system for execution. The incarnation of execution environments is implemented through modifications of this shell script which lead to minor extensions of UNICORE's incarnation logic.

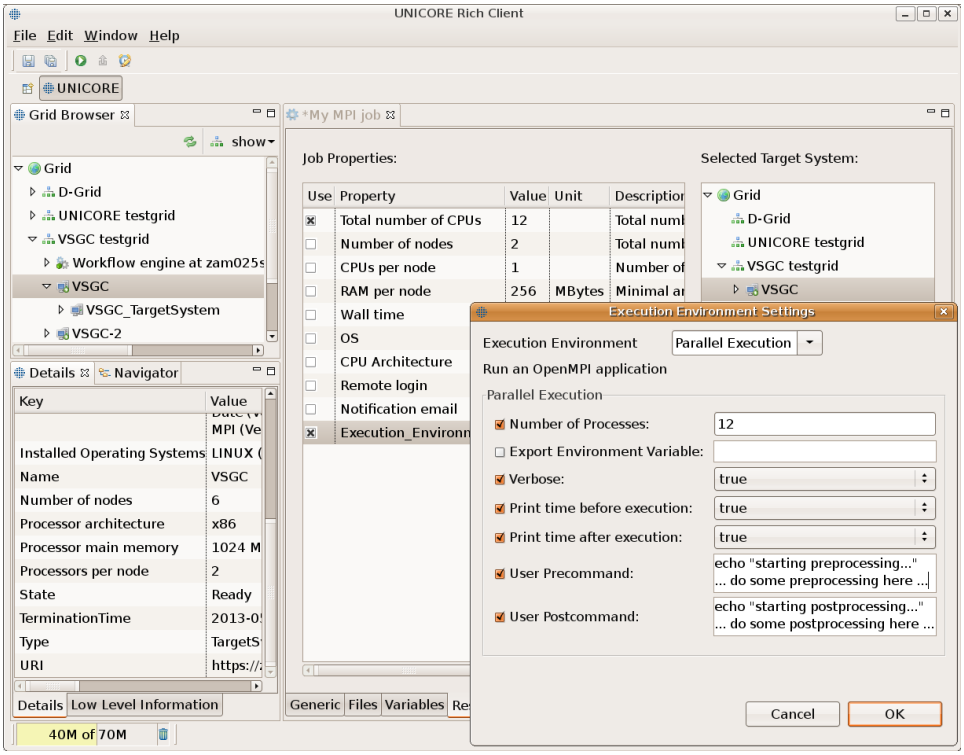

Figure 1. Dynamically created graphical user interface for selecting and configuring execution environments.

5 Extensions to the UNICORE Rich Client

UNICORE's graphical client suite, the Eclipse-based UNICORE Rich Client (URC) has been extended to let users select and configure execution environments (see Figure 1). To this end, the aforementioned resource property that is published by the UNICORE 6 server is parsed and the contained execution environment descriptions are extracted. The job editor of the URC has been extended with a GUI that displays available execution environments for selection. Once the user chooses an environment, the user interface dynamically adapts to the selection and shows all parameters, options, pre- and postcommands for configuration. The user's choices are saved inside the job description and the job can be submitted. Upon submission, the UNICORE 6 server extracts the execution environment configuration from the job description and modifies the incarnated executable shell script accordingly.

The example screenshot shows the dynamically created GUI for the parallel execution environment example as defined in Listing 2. Note how the different arguments, options, and pre-/postcommands are represented by different widgets. If the execution environment definition provides a *ValidValue* element, the user input is also fully validated and invalid input will lead to clear and detailed error messages. Also note the text fields for entering additional *user pre-/postcommands* which can hold arbitrary shell scripts if necessary.

6 Concluding Remarks

In the scope of this work, a new mechanism has been added to the job incarnation process of the UNICORE 6 Grid middleware. It has proven a helpful tool in the context of parallel application execution and in dynamically loading required DEISA software modules, the two use cases driving our development. The mechanism has been designed for flexibility and will certainly be employed to cover additional use cases. It allows for comprehensive, fine grained, graphical configuration of application execution environments, thus greatly elevating the experience of end-users with demanding applications. The developed techniques for automatic generation of graphical user interfaces from parameter annotations, taking into account both parameter typing and valid value ranges, can be applied in other scenarios as well. Since URC version 6.3.0, they are also used to build an adaptable main panel for the job editor when no tailored user interface exists for a given Grid application. Similarly to execution environment parameters, application parameters can be annotated with types, descriptions, and valid value ranges by administrators inside the IDB. Again, this information is published by the UNICORE server and evaluated by the client in order to build a user interface for parameterising an application. As a result, intuitive GUIs are immediately available for newly deployed applications in UNICORE Grids without any additional programming effort.

References

1. A. Streit et al., UNICORE - A European Grid Technology, High Speed and Large Scale Scientific Computing, Advances in Parallel Computing **Vol. 18**, 157–173, 2010, `http://www.unicore.eu`.
2. T. Bray, J. Paoli, C. Sperberg-McQueen, Extensible Markup Language (XML) 1.0, `http://www.w3.org/TR/1998/REC-xml-19980210`, 1998.
3. M. Snir et al., MPI: The Complete Reference, Vol. 1, The MPI Core, The MIT Press, 1998.
4. H. Lederer et al., DEISA: Enabling Cooperative Extreme Computing in Europe, NIC Series **Vol. 38**, 689–696, 2007.
5. E. Gabriel et al., Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation, LNCS **Vol. 3241**, 353–377, 2004.
6. W. Gropp, MPICH2: A New Start for MPI Implementations, LNCS **Vol. 2474**, 37–42, 2002.
7. G. Staples, TORQUE resource manager, Proceedings of the 2006 ACM/IEEE conference on Supercomputing, ISBN: 0-7695-2700-0, 2006.
8. I. Foster et al., Modeling and Managing State in Distributed Systems: The Role of OGSI and WSRF, Proceedings of the IEEE **Vol. 93, Issue 3**, 604–612, 2005.
9. A. Anjomshoaa, F. Brisard, M. Drescher, D. Fellows, A. Ly, S. McGough, D. Pulsipher, and A. Savva, Job Submission Description Language (JSDL) Specification, Version 1.0, Open Grid Forum, GFD, 2005.

Running License Protected Applications with SmartLM and UNICORE

Hassan Rasheed[1], **Angela Rumpl**[1], **Wolfgang Ziegler**[1]
Björn Hagemeier[2], **Daniel Mallmann**[2], **and Henning Eickenbusch**[3]

[1] Fraunhofer Institute, SCAI,
Schloss Birlinghoven, D-53754 Sankt Augustin, Germany
E-mail: {hassan.rasheed, angela.rumpl, wolfgang.ziegler}@scai.fraunhofer.de

[2] Jülich Supercomputing Centre,
Research Centre Jülich, 52425 Jülich, Germany
E-mail: {b.hagemeier, d.mallmann}@fz-juelich.de

[3] ANSYS Germany GmbH,
Staudenfeldweg 12, 83624 Otterfing, Germany
E-mail: Henning.Eickenbusch@ansys.com

SmartLM provides a generic and flexible licensing virtualization technology based on standards for new service-oriented business models. The software licenses are implemented as Grid services, providing platform-independent access just like other Grid resources and being accessible from resources outside organizational boundaries. The License Service is responsible for creating licenses based on the user requirements. These licenses are governed as Service Level Agreements based on evolving standards, and describes all specific conditions of the application usage like license owner, issuer, validity period, usage records, pricing, features, wall time, etc. The Orchestration Service co-allocates licenses and computing resources both in time and space, assuring that these resources are available at the same time for the execution of application. The Orchestrator retrieves license token from the License Service and transport it to selected computational resources while being sure that the license required is available to the application upon start-up and during execution. Secure agreements are used to transport license tokens through the Grid to computing resources. The license and compute agreements and conditions of use for an application are reached through negotiation between service providers and service customers. The Orchestrator integration with UNICORE is realized through UNICORE Integration Service that provides WS-Agreement based interface to interact with UNICORE servers. The orchestrator client, an extension of UNICORE rich client, is used to steer the license and computing resources negotiation, computing resources selection, data staging, job submission and execution management.

1 SmartLM Overview

The existing licensing models for commercial applications are focusing on software used on compute resources within an administrative domain. Licenses are provided on the basis of named user, node-locked host, number of concurrent jobs and are bound to hardware within the domain of the user and do not allow access from outside thus enforcing only local use of the protected applications.

SmartLM approach consists in treating and implementing software licenses as Web Service resources, and is based on standards for new service-oriented business models, thus providing platform and location independent access just like other virtualized resources. SmartLM solution is able to adapt itself to distributed environments and works with loosely coupled systems, because the software licenses itself, as Web service resources, become

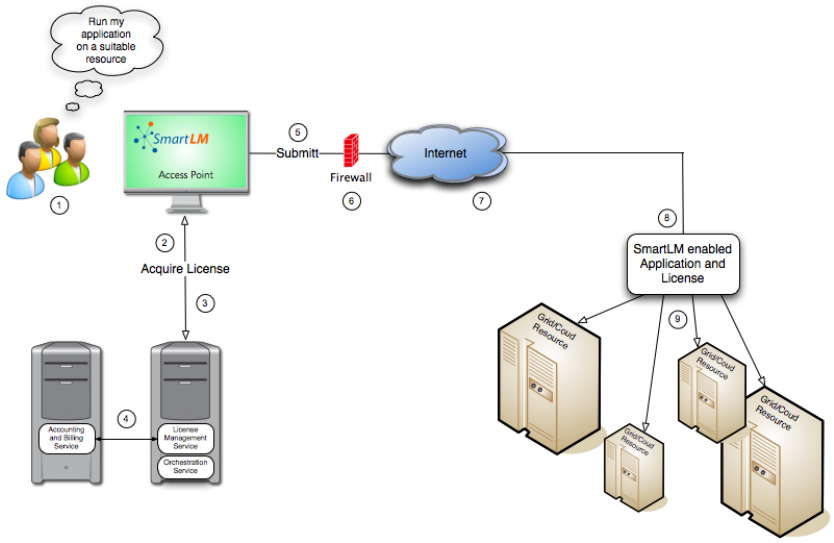

Figure 1. High-level view of the SmartLM architecture

loosely coupled as regards the issuer service and really dynamic also. The licenses are enveloped by Service Level Agreements, based on the WS-Agreement[1] and their management is based on tokens which grant the required flexibility in terms of time and space shifts. Token enforcement API is the interface between the applications and SmartLM license system. An application has to check the license tokens for correctness and verify if it provides sufficient features for the application to run in a certain configuration. The applications can run locally, with and without Grid middleware, or remotely, using Grid resources or resources in Clouds, even without network connectivity during the application run.

The high level view of the SmartLM architecture is shown in Figure 1. The user intends to have its license protected application executed on a suitable resource. The user negotiates with the License Service availability, terms of the license and the price. The license agreed upon is reserved and a usage record with the license usage information is created. The usage record is retrieved by the accounting and billing system. The license token is scheduled to the remote resource (eventually together with the application). The token traverses the firewall which protects the site of the user. The token is traveling through the Internet. To avoid exhibiting the content to unauthorized parties it has been encrypted after creation. SmartLM enabled application together with the license token for the execution is moved to the selected remote resources. The application checks the license token to validate the authorization of the application execution.

2 License Service

The License Service is the core service which follows a layered architecture comprising 6 layers: Coallocation, Authentication, Administration, Management, Business, and Per-

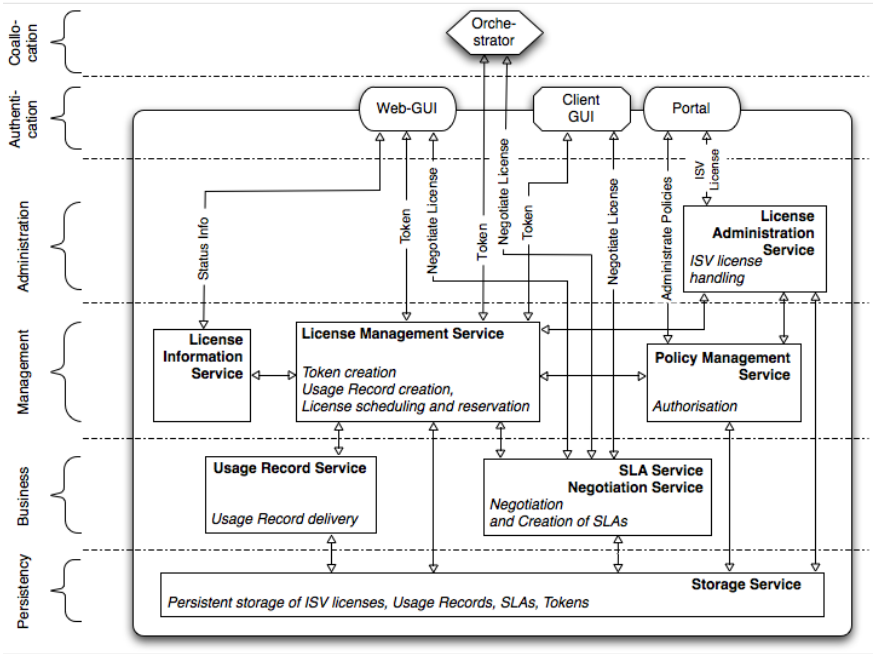

Figure 2. Layered architecture of the License Service

sistency. Figure 2 depicts the major communication paths between the components. The Authentication Layer comprises three components, which allow the user to authenticate himself and - after successful authentication - to access the core services of the License Service. In the Administration Layer, the License Administration Service allows including new licenses, removing licenses, and adding features to already installed licenses.

The Management Layer comprises three core services: The License Management Service is responsible for the creation of license tokens taking into account the authorization of the user (signing and encryption of the token), creation of the initial and updated version of the Usage Record based on the actual license usage, and scheduling and reservation of licenses. The License Information Service provides detailed information on existing, the actual use of licenses and license reservations. The Policy Management Service is the central entity for managing end evaluating policies that define the access to and the usage rights of a certain license or license feature for a user or a group of users.

The Business Layer hosts two services: the Usage Record Service manages the Usage Records received from the License Management Service. It notifies the external Accounting and Billing Service that records are available for further processing. The SLA and Negotiation Service is a crucial component of the License Service. It is responsible for creating the agreements for using licenses based on the requirements of the user and the availability of licenses for this user. Moreover, the Negotiation Service allows negotiating the terms of a license, especially the time when a license will be available for the user. While processing a license request the service also communicates with the external Ac-

counting and Billing Service for retrieving a price for the requested license and the budget already used by this user or user group. And finally the Persistency Layer provides the Storage Service that stores arbitrary XML data on disk.

The License Service is capable of managing all licenses that support the new licensing mechanism owned by a company, thus providing a single point for license management and an easy and comfortable access to the entire license information. Moreover it supports the negotiation of the license terms to be used for running an application. These licenses are governed as Service Level Agreements and describe all specific conditions of the application usage like license owner, issuer, validity period, usage records, pricing, features, wall time, etc.

3 Orchestration Service

The Orchestration Service co-allocates licenses and computing resources both in time and space. It is responsible for assuring that licenses and computational resources for the execution of applications are available at the same time. It allows the use of remote Grid resources for the execution of an application which needs a software license at run time, while being sure that the license required is available at the remote site when the application will start-up and during execution.

The Orchestration Service interacts with the License Service and the UNICORE Integration Service for common time slot negotiation based on computational resources and licenses availability. If successful, the negotiation results in both the reservation of computational resources and the licenses required for the application execution. Service Level Agreements are used to describe the reservations and the corresponding guarantees. A step-by-step description of the Orchestration Service interaction with other services for co-allocation and SLA creation process is shown in Figure 3.

In short, the client requests template for computational and license resources where the user specifies the name of application and license for his submission. The Orchestration Service returns template with chosen application and available features based on the user authorization after contacting the License Service. Now the client proceeds by sending negotiation quote with selected application features, license duration and proposed time slot for application execution. The Orchestration Service negotiates with both services and received templates back on their availability within client specified time frame. The co-allocator checks whether the license and computational resources are available for a common time slot.

In case of successful negotiation, the Orchestration Service delivers the consolidated templates into a single one to the client for rejection or confirmation. In case the client is comfortable with the negotiation results he creates a single SLA with the Orchestration Service, which in turn creates SLAs with the License Service and the UNICORE Integration Service. The License Service in turn creates the token and embeds it into the SLA created with the Orchestration Service. Upon successfully creating the individual SLAs with both services, the Orchestration Service finalities the SLA with the client by sending the Endpoint Reference of the SLA to the client. The Orchestration Service extracts the token and transfer it together with the user's job to the remote site, to a place where the application later accesses the token to authorize the execution.

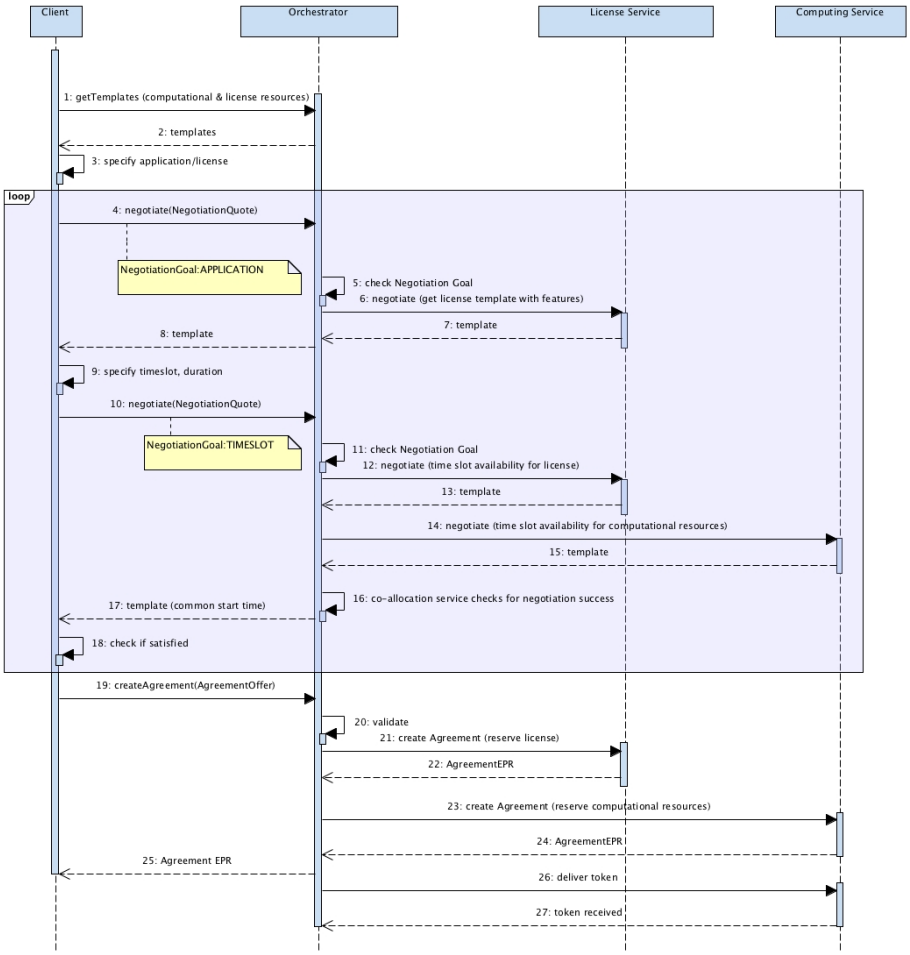

Figure 3. Co-allocation and SLA creation

4 UNICORE Integration Service

The Orchestration Service rely on UNICORE Grid middleware[2] to deal with the execution environment. The Orchestrator integration with UNICORE is realized through UNICORE Integration Service that provides WS-Agreement based interface to UNICORE environment. Since the Orchestration Service uses WS-Agreement and WS-Agreement-Negotiation as a standard interface to external components allows to easily integrate the Orchestration Service in different environments without having to modify it for each new environment. Therefore, the UNICORE Integration Service provides WS-Agreement based templates for resource reservation, data staging, job submission, job monitoring and execution management. In SmartLM, this service has been extended for the transfer of license token directly into job working directory of the submitted application during SLA creation.

5 Orchestrator Client

The Orchestrator client is an extended version of the eclipse rich client, which is available as client for UNICORE. It further extends the graphical user interface to the Orchestration Service to integrate the user interface elements for steering the license and computing resources negotiation, computing resources selection, data staging and job submission and management. The client uses application GridBean for each SmartLM enabled application. The application GridBean is a concept from the Grid Programming Environment (GPE)[3] that provdies graphical user interfaces for generating job description and input/output files. Sicne the Orchestrator client communicates with the Orchestration Service over WS-Agreement, it cannot directly interact with Unicore servers for job submission and data staging. Instead the Unicore Integration Service perform these operations.

6 ANSYS CFX Example

ANSYS has integrated SmartLM licensing solution in ANSYS CFX[4] application. This application is a high-performance, general purpose CFD program that has been applied to solve wide-ranging fluid flow problems. We have successfully demonstrated the usage of ANSYS CFX on UNICORE resources through the Orchestration Service.

Acknowledgement

This work has been partially funded by the European Commission in the 7th Framework Programme project SmartLM under grant agreement 216579. The grant period is from 01.02.2008 until 31.07.2010.

References

1. http://www.ogf.org/documents/GFD.107.pdf
2. http://www.unicore.eu
3. http://gpe4gtk.sourceforge.net/GPE-Whitepaper.pdf
4. http://www.ansys.com/products/fluid-dynamics/cfx

Resource Usage Accounting for UNICORE

Piotr Bała[1,2], **Krzysztof Benedyczak**[1,2], **and Marcin Lewandowski**[1,2]

[1] Interdisciplinary Center for Mathematical and Computational Modelling,
University of Warsaw, Warsaw, Poland
[2] Nicolaus Copernicus University
Poland
E-mail: {bala, golbi, ml054}@mat.umk.pl

In this paper we cover recent developments and future plans in an area of resource usage accounting in the UNICORE grid. The development aims to maintain Polish National Grid Infrastructure and is performed within the PL-Grid project. The developed accounting system collects and merges data from the XNJS and a batch system. The architecture is open and pluggable so sensors for yet unsupported batch systems can be easily integrated. In this work we employe the OGF Usage Record specification, as the data format ensuring interoperability. We also provide an evaluation of the Resource Usage Service, which is available as an OGF draft specification. Finally the paper describes an integration with the PL-Grid central accounting system and an additional support for aggregated usage records.

1 Introduction

Accounting of a resource usage is required in production grids due to two main reasons. Scientific supercomputer centres, which devote computational resources to the grid, need information on actual usage of their hardware. This data is required for the reporting to supervising bodies (such as a government or funding agency), and to perform analysis which resources are the most crucial, overloaded or underutilized.

Another aspect of the resource usage accounting is monitoring of shares which are provided to particular users or groups of users. This issue is similar to the functionality of a cluster batch queuing system but is performed on the grid level. Users which are granted limited amount of resources should not consume more of them, while particular resources should be available for the groups working on the high priority or time constrained scientific challenges. Grid economics and related billing systems fill into this category, however in the academia this approach is still rarely adopted. On the other hand we can observe increasing effort to converge grid and cloud computing and to make this happen accounting and billing are essential.

In addition to the described above main purposes of a grid accounting, grid administrators can use accounting data on an everyday basis. High accounting records can be used to select suspected users whose actions should be more carefully monitored (for example to check if resources are used in accordance with rules and obligations). Resource usage statistics can be also used to tune resources distribution to achieve better resources utilization in the grid.

In this paper we present our recent developments and future plans in the area of resource usage accounting. The development is performed within the PL-Grid[1] project which maintains the Polish National Grid Infrastructure. The following section presents relevant standards in the area of grid accounting. The main part is devoted to the architecture of the

accounting system along with an information on data flow in the system. Comparison with selected existing solutions is presented as well.

2 Motivation and aim

The PL-Grid project requires resource usage accounting for numerous purposes. The most important are:

- Possibility to provide various indicators for project reporting.
- Possibility to enforce the concept of computing grants. Computing grants are assigned to users by the project's committee and contain maximum amount of resources that a user can consume. The grant concept allows for fair resources allocation and enables possibility of prioritization of the research conducted on the shared resources.

The main driving force to create a new accounting solution, is lack of existing open source accounting system for the UNICORE software. Some efforts were undertaken in the past to create accounting or billing systems, however none of them is both open source and complete. More detailed evaluation of existing systems is presented in the section 5. For the readers unfamiliar with the UNICORE grid middleware a good introduction can be found for instance here[2] and on the project's website[3]. Because of the limited space we will not discuss them here. During the planning of the system there were some additional factors that we had to take into account. First of all the PL-Grid project's grid infrastructure is heterogeneous. gLite middleware[4] is installed next to UNICORE, providing access to the same machines. Other middlewares are under consideration for deployment and may be used in the future. Next, some of the HPC sites do not have any experience with the UNICORE middleware and developed solutions have to be easy to install and maintain. Finally the PL-Grid project deploys a central accounting database, which must be populated with a data from all middlewares installed - an interoperability cannot be neglected.

Based on this, set of required features of the accounting system for UNICORE was defined as follows:

- The system must be able to collect all resource usage data of jobs submitted via UNICORE.
- It must not interfere with other middlewares and with ability of a direct access to a resource management system.
- It must be easy to install and maintain.
- It must be able to easily export the data to third party accounting solutions.

As we are aware that there are many solutions for presenting accounting records (usually web based) we decided to concentrate our effort on providing a good resource usage acquisition. Therefore the query part of the system which presents collected records is minimal; we provide only basic tools which should be sufficient for administrators and testing.

3 Related standardization activities

As it was noted in the previous section interoperability with other accounting solutions is very important. To achieve it we have considered several standardization efforts and decided to adopt most of them.

We are aware of three important standardization activities, all of them under the wings of Open Grid Forum[5] (OGF). Unfortunately only one of them resulted in creation of a formal document: the *Usage Record — Format Recommendation*[6] which can be used as guidelines. Usage Record specification defines XML schema of a single, fine-grained piece of accounting data for a grid job in a particular point of time. Therefore a singe job can possess multiple URs associated with it, each of them describing a job's state in consecutive stages of its life cycle. The UR specification gains popularity, and is being used in other accounting systems. While current deployments shows some deficiencies (as lack of a field for storing a virtual organization to which a job belongs), the standard is extensible and we used it as a basic data format of our system.

Besides of the Usage Record format there is an accompanying format named *Aggregate Usage Representation*[7] (AUR). The AUR format is currently in the draft state, however we have found it valuable. Therefore we have decided to use it as a guidance for creation of a format used for coarse-grained usage records (i.e. providing usage statistics for an entity in a period of time). The AUR format is strictly based on the UR format and defines which fields should provide summarized data.

The other group of standardization activities is related to the definition of interface for managing usage data. The proposed standard is named *Resource Usage Service*[8] (RUS) and is in a quite early stage. We must admit that it is hard to consider this work as a real recommendation, and it is difficult even to assess it. It is due to fact that this is still ongoing work and subsequent versions of the document contain loads of significant differences.

The RUS specification assumes usage of the UR specification as its fundamental data format. The RUS specification uses Web Services as a communication technology. This is one popular option in the accounting field, while the another one is usage of Message Oriented Middleware which is better suited for fast and reliable transportation of usage records. The Web Services approach provides a better querying capabilities.

Evaluation of the Web Service RUS interface determined two important problems. The first one is related to the base query operation of the interface which uses XPath language[9] for selecting interesting records. This operation brings authorization problems, as access to it can be granted only for users having full read permissions to all records. It is hard to provide efficient implementation which allows ordinary users to check how many resources were consumed. The second problem is that when using XPath queries it's hard to provide efficient caching and indexing of common selection criteria, which are in case of accounting quite easy to foreseen.

The conclude design directions, we have decided to use most of the RUS interface but at the same time additional operations will be added to overcome the spotted problems. Moreover, we have decided to create the network access layer pluggable so other (even absolutely different) communication paradigms can be easily integrated.

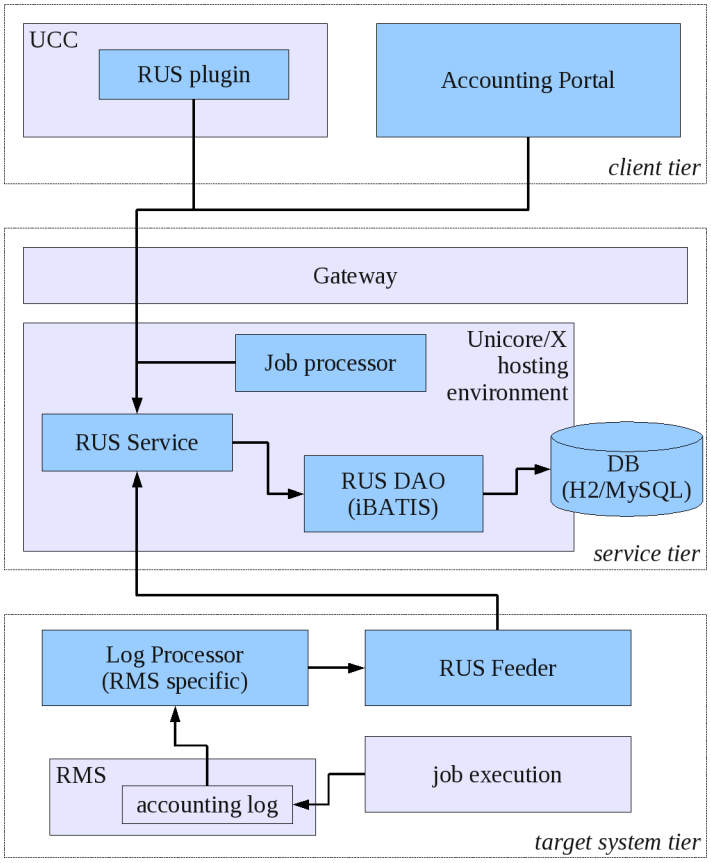

Figure 1. The three tiers of the UNICORE accounting system. The accounting components are highlighted and their interconnections are marked.

4 The UNICORE RUS system

The main difficulty in development of the UNICORE accounting system is the requisite to gather data from the two sources: the resource management system and the XNJS which is UNICORE component handling job's invocation. This problem results from the fact that grid accounting requires both the grid data (as a grid name of a user, virtual organization etc.) and the fine-grained usage data (as amount of consumed memory), which is only available on a Resource Management System (RMS) level. We can note here that a similar problem occurs in other popular middlewares as well.

The components of our system are presented in the figure 4. We can divide them into three tiers. The first one is the client tier. As it was noted in the introduction, implementation of this layer contains only a minimal set of tools. We will provide plugin for the UNICORE Command Line Client which should be useful for administrators and for testing. We plan to create a simple web application capable of presenting the most fundamental accounting data as well.

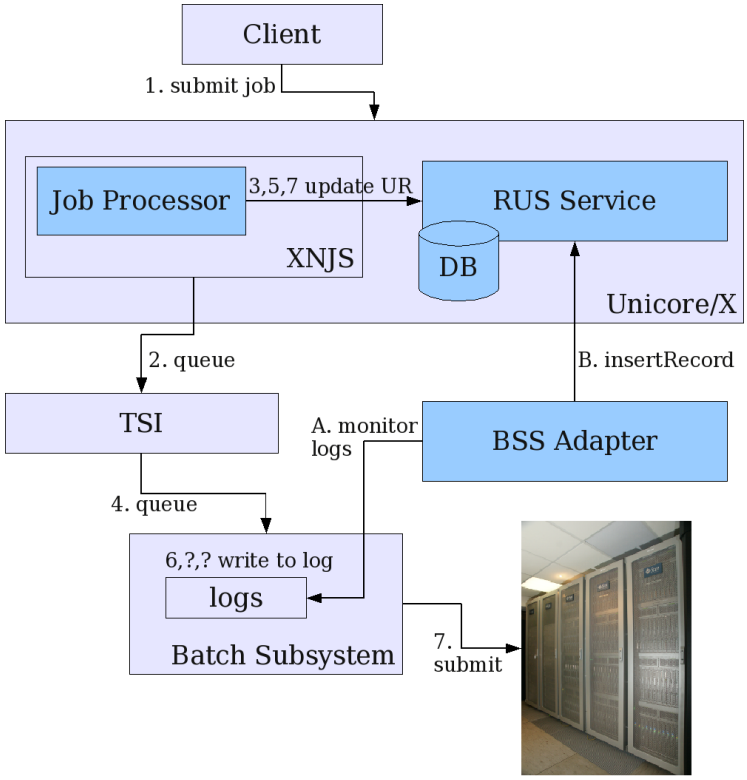

Figure 2. The picture depicts the two flows of external events which influence the operation of the accounting system. The job submission constitutes the first flow (denoted with numbers), while RMS maintenance of a job makes up the second (denoted with capital letters).

The next — service — tier is the heart of the system. It is composed out of the three main modules:

- The accounting database which holds Usage Records.
- The RUS service which provides a general purpose network interface to the system. The implemented operations allows for both querying the stored data and for managing (i.e. inserting, deleteing and updateing) it.
- The XNJS Job Processing plugin which collects the grid jobs' data.

The last, target system tier, collects detailed usage records from the site Resource Management System (RMS). The data is then sent to the RUS service. There are two components in this tier. The first one is the universal daemon which controls the RMS log monitoring process and acts as a RUS service client. The second component is a thin, RMS-specific parser module.

As it was mentioned above the system must collect data from two sources: XNJS processing pipeline and RMS logs. The events flow is presented in the figure 4. As we can

see the accounting records linked with a particular job are inserted to the database after each change of the job's status, i.e. after job is queued, started or finished. The RMS log monitoring component periodically scans the accounting logs and pushes the newly added records to the RUS service. The service is using a special record matching logic to merge data coming from the two sources. The algorithm is flexible and allows for an incomplete record data, coming in an arbitrary order. This feature eliminates race conditions, which could otherwise easily occur in numerous cases.

The accounting system provides the two extension points. The first one (described above) is the possibility to add thin adapters for various RMS. Reference implementation for Torque RMS is already provided, we also plan to add the Sun Grid Engine adapter. The second extension point is currently being implemented. It allows for export of accounting records in arbitrary formats using other then Web Services transport technologies. We are createing the extension, which will send accounting records with internal PL-Grid format using ActiveMQ. This extension will be used in the PL-Grid deployment.

Security of the system is composed of the typical UNICORE components. Authentication between RUS service and clients (both the end user and the target system feeder) is performed using the client-authenticated TLS. Authorization is managed with XACML policies and we provide standard entries for the RUS service. In the future we also plan to investigate usage of common authorization framework for other means of network access.

5 Comparison with other solutions

There are several existing accounting solutions available for the Grid. As it was mentioned before, UNICORE related systems are either closed or not complete. We was able to find the following ones:

- The RUS service available at the UNICORE SourceForge website. The code is merely a stub of a real implementation: only one query operation is implemented and it is incomplete. The service assumes that accounting data is stored in an XML file and this file is simple returned. It's obvious that this tool can not be used in practice. This system is a part of OMII-Europe effort on standardized accounting and since project is finished there is no information avaiable.

- System developed in United Institute of Informatics Problems, Belarus for the Belarusian National Grid Network[10]. At the moment of preparing this paper, information on this system are very limited (regular paper still did not appear). Anyway the system is closed as it was confirmed by the authors in private communication. Therefore it is not possible to use this system in open science world and obviously it is hard to compare our approach with it. The Belarusan system is not standards based and it is not open for interoperability, however it is providing billing capabilities.

- There are some pieces of information in the Internet on the DEISA accounting solutions for the UNICORE, however only concrete data is available on DART[11] — a simple user front-end for presentation of an accounting data in a text form.

Other grid middlewares provide a sophisticated and full-fledged accounting systems. The main players here are gLite APEL, SGAS and DGAS. Quite comprehensive review

of those systems is presented in the paper[12]. We can see that systems can be divided into two groups: those using messaging (MOM) as transport paradigm and those employing Web Services for this purpose. While our system falls into the later category we provide extensibility point which allows for exporting data to other systems, and at least one MOM exporter will be available.

6 Summary

In this work we present accounting solution for the UNICORE middleware developed within the PL-Grid project. As the solution is being adopted in the heterogeneous grid environment, we trust that it is a perfect place to evaluate interoperability and standards available.

The shortcomings of the existing standards lead us to the following conclusions: (a) network interface standards should be finished, (b) Usage Record specification is generally useful, however should be slightly extended to cover also Virtual Organisations and other missing concepts and (c) standards for MOM based transports mechanism should be defined. The last point is fairly trivial as MOM does not require a complicated interface definition (as Web Services does) and general implementation guidelines are sufficient.

Technical outcome of our work is open and standards based. While we ensure easy installation and effortless maintenance we hope that our system is generally useful and provides valuable contribution to the UNICORE community.

This work was supported by the Polish Infrastructure for Information Science Support in the European Research Space PL-Grid project, contract number: POIG.02.03.00-00-007/08-00.

References

1. The PL-Grid Project website. `http://www.plgrid.pl` (accessed on 13.06.2010)
2. A. Streit: UNICORE: Getting to the heart of Grid technologies. eStrategies | Projects, 9th edition, British Publishers Ltd, Mar. 2009.
3. The UNICORE project website. `http://www.unicore.eu` (accessed on 13.06.2010)
4. The gLite project website. `http://glite.web.cern.ch/glite/` (accessed on 13.06.2010)
5. Open Grid Forum organization website. `http://www.ogf.org` (accessed on 13.06.2010)
6. R. Mach, R. Lepro-Metz, S. Jackson: Usage Record. Open Grid Forum format recommendation (2006-2007). `http://www.ogf.org/documents/GFD.98.pdf` (accessed on 28.03.2010)
7. R. M. Piro, P. Canal, J. Gordon, D. Kant, A. Khan, X Chen (ed.): Aggregate Usage Representation Recommendation (Version 1.0, draft). Open Grid Forum, December 2006.
8. X. Chen, A. Khan, J. Ainsworth, S. Newhouse, J. MacLaren: WS-I Resource Usage Service (RUS) Core Specification (draft 19).

http://forge.gridforum.org/sf/projects/rus-wg (accessed on 28.03.2010)
9. J. Clark, S. DeRose (eds.): XML Path Language (XPath), Version 1.0. W3C Recommendation 16 November 1999. http://www.w3.org/TR/1999/REC-xpath-19991116 (accessed on 13.06.2010)
10. A. Lukoshko, A. Sokol: UNICORE-related Projects for Deploying the Belarusian National Grid Network. UNICORE Summit 2009 presentation (paper not yet published). http://www.unicore.eu/summit/2009/presentations/08_Lukoshko_Belarus.pdf (accessed on 12.06.2010)
11. DART tool documentation. http://www.deisa.eu/usersupport/user-documentation/deisa-accounting-report-tool (accessed on 12.06.2010)
12. M. A. Pettipher, A. Khan, T.W. Robinson, X. Chan: Review of Accounting and Usage Monitoring. The University of Manchester, July 2007.

The Key Role of the UNICORE Technology in European Distributed Computing Infrastructures Supporting e-Science Applications in the Decades to Come

Morris Riedel[1], Mohammad Shahbaz Memon[1], Ahmed Shiraz Memon[1], Sonja Holl[1], Daniel Mallmann[1], Nadine Lamla[1], Achim Streit[1], and Thomas Lipppert[1]

Jülich Supercomputing Centre, Institute for Advanced Simulation,
Research Centre Jülich, 52425 Jülich, Germany
E-mail: m.riedel@fz-juelich.de

Computational simulations and thus scientific computing is the third pillar alongside theory and experiment in science for a few decades now. In the last decade, the term Grid and later e-science evolved as a new research field that both focus on collaboration in key areas of science using so-called next generation distributed computing infrastructures (DCIs) to extend the potential of scientific computing. During the past years, significant international and broader interdisciplinary e-research is increasingly carried out by global collaborations that often share the resources of these DCIs. Examples of these DCIs have been the Enabling Grid for e-Science (EGEE) infrastructure as well as the Distributed European Infrastructure for Supercomputing Applications (DEISA) in Europe. Despite of the early success we can observe a change in the near future with these infrastructures since these project-based funded infrastructures like EGEE or DEISA move towards a more sustainable funding model in Europe. The result is a different landscape of DCIs as experienced in the last decade leading to the European Grid Initiative (EGI) infrastructure and the Partnership for Advanced Computing in Europe (PRACE) infrastructure. At the same time end-user communities organized themselves better than before and developed roadmaps for their scientific endeavours. In this context, the European Strategy Forum on Research Infrastructures (ESFRI) released an interesting roadmap of 44 projects that bear the potential of a high amount of end-users that also require DCIs for their scientific e-research. Another promising roadmap alignment of DCI end-users can be observed in the context of the Virtual Physiological Human (VPH) to push and focus efforts related to e-Health. In parallel to all the aforementioned 'emerging changes' UNICORE is still established as the technology of choice for resources used in High Performance Computing (HPC). The question remains whether this will be the case in the future and to which extend new e-research endeavours will influence the funding opportunities (and thus major developments) for the UNICORE community. This contribution will give some answers to these questions by providing one potential UNICORE roadmap that reveals a promising future.

1 Introduction

The Grid computing vision as initially defined in 1998 by Foster et al.[2] has been established in the last decade leading to a wide variety of different applications, tools, and Grid infrastructures. Although we learned that the vision of *'one single infrastructure like the power Grid'* is not easy to implement in computer science the term Grid evolved. In fact, a better understanding of this vision revealed many challenges to realize this vision. First and foremost, like the *'power Grid'* the power must be provided by some entity and thus we have to consider also the *'power plants'* behind the originally coined idea that *'power simply comes out of the power outlet'*. Hence, there must be computational resources available in a reliable and sustainable way in order to provide computing power.

Another learning experience brought us to the complexity of computing services that have by far a broader range than *'pure electrical power'*. We experience this by the plethora of different services available on different distributed computing infrastructures (DCI) today. And the fact that there are different DCIs shade the Grid vision as well with challenges equally existing in 'power Grids' in terms of requiring different adapters to use the different 'power infrastructures' existing in different countries. Hence, in order to use another DCI as the commonly used one, adapters are in the majority of the cases still necessary in order to enable production runs by scientists.

More precisely, DCIs such as NorduGrid, EGEE, DEISA, or TeraGrid provide a wide variety of different Grid services to end-users on a daily basis today. But this era of distributed computing infrastructures was fundamentally driven by supporting dedicated Grid technologies per infrastructure (e.g. ARC in NorduGrid, gLite in EGEE, Globus in TeraGrid, and UNICORE in DEISA). This can be at least particularly explained by the lack of open standards in the field that would make it possible for end-users to choose the technology they desire. Also, dedicated technologies exist, because collaboration among those technology providers just recently become more fruitful and have been before only established in a pair-wise fashion.

As a consequence it is very difficult to realize Grid applications supported by the Grid vision initially coined by Foster et al.. To provide an example, we still strugle to create applications that particularly would benefit of accessing resources that are part of different DCIs (i.e. HTC vs. HPC-driven) and well-known data-repositories (e.g. LHC data in EGEE/EGI) today. Hence, each of the DCIs defines clear boundaries (i.e. middleware technology, access policy, etc.) that are difficult to cross in order to realize new innovative types of distributed computing applications. More notably, the usage of one single DCI by one end-user is still a constant struggle (accounts, certificates, grants to get time, etc.) and when crossing the borders of DCIs for applications it becomes nearly unmanageable today.

But the near to med-term future will experience a fundamental paradigm-shift that promotes the Grid vision much better as it can be observed today leading to a much more intensive infrastructures usage. The uptake of the Grid computing concept in general and its implied distributed computing methods in particular will be majorly pushed by the evolution of emerging interoperable networks of standards-based technologies. The first major change is that the specialization of the infrastructures will become much more flexible by the broader usage of emerging open standards within their dedicated Grid technologies making it much easier for end-users to choose their technology of choice.

A closer look in Europe reveals, for instance, that the transition process between EGEE and EGI will lead us to an era where there will be multiple Grid technologies deployed thus lowering the boundaries as present today. Similiar changes to avoid vendor-locks whereever possible are expected to appear in the next TeraGrid phase as well being funded under the extreme digital scheme. All these transition processes bear chances for the UNICORE technology to extend its amount of deployments while there is no doubt that many sites will still stick to their previously used technologies when they see no benefit in moving to other solutions.

The second major change is that end-user communities become more organized as experienced previously. Supporting independent single scientists with their preferred tools and technologies has been in many cases cumbersome since they have often mutually exclusive demands (e.g. grants vs. ad-hoc access, broker approach vs. explicit choose of sites, trivial security demands vs. strong security demands, etc.) or simply require too much technology solutions to 'provide them all at once'. We will explore in this contribution different roadmaps of whole organized end-user communities that are different and comparable to the strong end-user community that evolved around the Large Hadron Collider (LHC) in physics. The difference is that these new evolving communities come from all different areas of scientific disciciplines and thus Grid computing will be not anymore majorly pushed only by the LHC requirements.

One of the key goals of the new emerging DCI landscape with EGI and PRACE will be to support these projects for decades to come in their data analysis process. The process in turn is very complex and in many cases scientific-field specific with data acquisition approaches using research vessels, telescopes, observatories, antennas, and other large instruments. We can expect that many of these projects will lead to high loads in data repositories that in turn require computational resources of DCIs in order to effectively analyze and efficiently process this data. While there are overlapping requirements with already existing end-users in the mentioned DCIs, we can also foresee that the DCIs have to embrace change in order to satisfy the requirements of new end-user communities. Also, we can expect several new forms of the usage and/or deployments of DCIs aligning them much more closer with scientific tools, community-specific data repositories, and large-scale instruments.

We thus experience a *'situation of change in the DCI landscaope'* and it is an interesting question what the potentials and perspectives for a Grid technology like UNICORE can be in the future. Therefore, the idea behind this paper is to put several emerging potentials for the UNICORE technology in a sequential order. Based on this, conclusions can be drawn that enable the creation of a long-term perspective for UNICORE for decades to come. The goal is thus to create one potential roadmap for the UNICORE technology with a particular focus on existing challenges that most likely will provide funding opportunities in the next decade.

The remainder of this paper is structured as follows. After the introduction into the problem domain of distributed computing infrastructures, Section 2 gives insights into e-science with a particular focus on the European landscape in general and the UNICORE technology in particular. Section 3 reveals certain aspects of how UNICORE can be understood as part of the European Middleware Initiative and provides insights into related future benefits. While Section 4 focusses on potentials for UNICORE in the context of the ESFRI roadmap, Section 5 describes further potential pathways for the UNICORE technology in the context of the VPH roadmap. Section 7 outlines other e-Research endeavours that will play a role in the med-term starting with Cloud computing over Green IT even towards future e-Business opportunities. Section 8 summarizes the findings of this paper as one potential UNICORE roadmap and provides some concluding remarks.

2 e-Science with UNICORE

The first building block of the roadmap starts with a general consideration of the basis of research endeavours in our field that have been collectively coined as 'e-science' following the idea given by the initial definition of J. Taylor[3]:*'e-science is about global collaboration in key areas of science and the next generation infrastructure that will enable it'*. Since this definition in 2005 we have seen many e-science applications using different incarnations of this next generation infrastructure such as DEISA and his DEISA Extreme Computing Initiative (DECI)[4]. There have been also numerous books like *'Workflows in e-science'*[5] or *'Design patterns for e-science'*[9]. Nevertheless, many of these research activities have just began to take an advantage out of these infrastructures and thus many grand challenges still remain unsolved (e.g. cancer, alzheimer, etc.). Hence, there is a requirement to fund these activities in the next years to decades in order to enhance the utilization of the infrastructures and to engage in the open or even new emerging grand challenges (e.g. continuation of flights during volcanic erruptions).

As mentioned in the introduction, the EC and other international funding organizations spent the last decade to create these so-called *e-infrastructures* and that have been used with initial e-science applications to tackle computational-intensive problems. Along with traditional approaches such as empirical, theoretical, and experimental studies, the more recent computational science and e-science approaches have intensively used such e-infrastructures in the last years. The aforementioned collaboration in key areas of science is majorly expressed through the creation of so-called *Virtual Research Communities (VRCs)* that bring the best brains in research together while using the services provided by these infrastructures. As indicated by K. Glinos at OGF28[8], the last calls in the EC Infrastructure Unit focussed on DCIs, but the future proposal calls will be oriented towards these virtual research communities and their necessary environments. More precisely, the next call 9 in this unit for Framework Programme (FP) 7, for example, explicitly mentions in one of its objective the support of these and related activities named as *'INFRA-2011-1.2.1:e-science environments'*.

These upcoming EU calls represent one building block of the roadmap since it provides the opportunity for the UNICORE community to continue or further extend the research endeavours undertaken in various EC-funding based research projects. Of course, the usage of UNICORE will continue through the DEISA infrastructure and potentially within PRACE as well and thus contribute to numerous e-science related research endeavours. But we have also seen in the past that UNICORE is a viable basis for many dedicated e-science-related research projects such as the OpenMolGrid[16], Chemomentum project[10], or activities in collaboration with the WISDOM community[6]. We can thus conclude that there is a high potential for the UNICORE community of being funded through one (or more) future e-science related research projects similiar to the one mentioned before. More precisely, UNICORE can be used to create scientific-field specific e-science environments through its extensibility or can be simply used by e-scientists in order to effectively access numerous infrastructures-based computational resources. We provide a classification of how UNICORE might be used to perform e-science in Riedel and Kranzlmueller et al.[1].

3 UNICORE as part of the European Middleware Initiative

The next building block of the roadmap outlines potential pathways via the European Middleware Initiative (EMI) project[18] that can be seen as a technology alliance of UNICORE, ARC, gLite and dCache. One vision of the EMI project is to create a software package that can be in its idea compared with the Microsoft Office suite. In a similiar manner, end-users may use gLite for data-intensive applications (like powerpoint for presentations), or may use ARC services for HTC-intensive computings (like Word for letters), or may use UNICORE for HPC-based applications (like Outlook for e-mails). Hence, the goal of this project is to provide a Grid technology suite that complements each of the Grid technologies with each others key features. The interoperability between them is based on improved standards based on OGSA-Basic Execution Service (BES)[7] and the Job Submission and Description Language (JSDL)[11] or the Storage Resource Manager (SRM)[13].

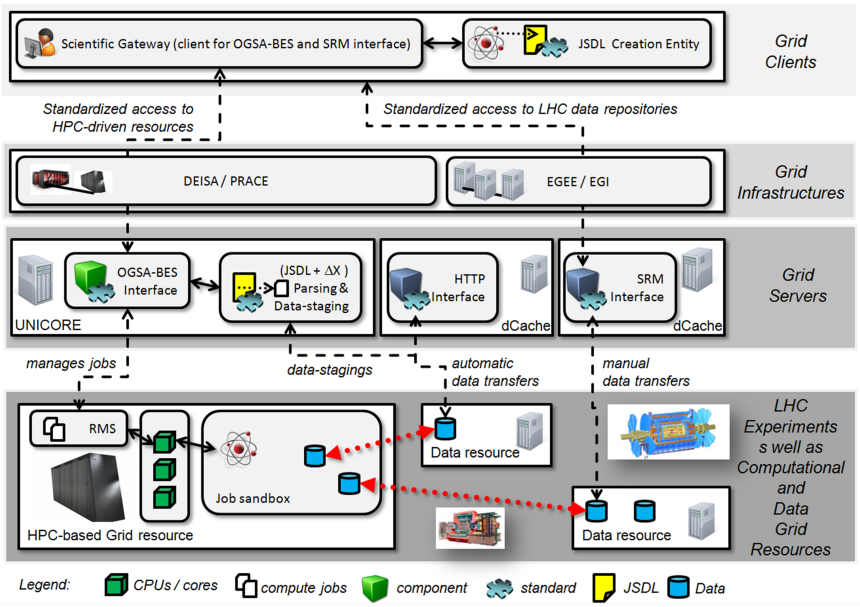

Figure 1. One goal of the EMI project is to enable the use of UNICORE with dCache in order to transfer LHC data to HPC-based infrastructures such as DEISA/PRACE thus enabling parallel computing analysis.

As shown in Figure 1, one of the major expected potentials for UNICORE as part of EMI is not only its technology evolution, but also the access to the EGEE/EGI data silos thus enabling the analysis of LHC data derived from its four different experiments that are conducted in the next decade. Furthermore, as UNICORE has been traditionally a rather HPC-oriented middleware it has been never deployed in EGEE, but as small to medium HPC resources are expected to become a part of EGI in the next years, UNICORE might become an option for a few sites that deploy the EMI software suite in the next years. This in turn would lead to more deployments in the future that is theoretically not bound to the amount of supercomputers available as we can observe in DEISA.

4 UNICORE and the ESFRI Roadmap

An important building block of the UNICORE roadmap is the roadmap released from the *European Strategy Forum on Research Infrastructures (ESFRI)*[12]. ESFRI in general and its roadmap in particular aims to support a coherent and strategy-led approach to organize pan-European and global research communities. The current 2008 roadmap lists 44 projects out of 7 different research areas and many of them have in common that costly resources (ships, antennas, telescopes, etc.) are involved that need to be shared across a geographically dispersed user community. Hence, similiar like the High Energy Physics (HEP) community around the LHC, these projects have become an organized set or clusters of communities that focus on specific areas of science as listed in Figure 2.

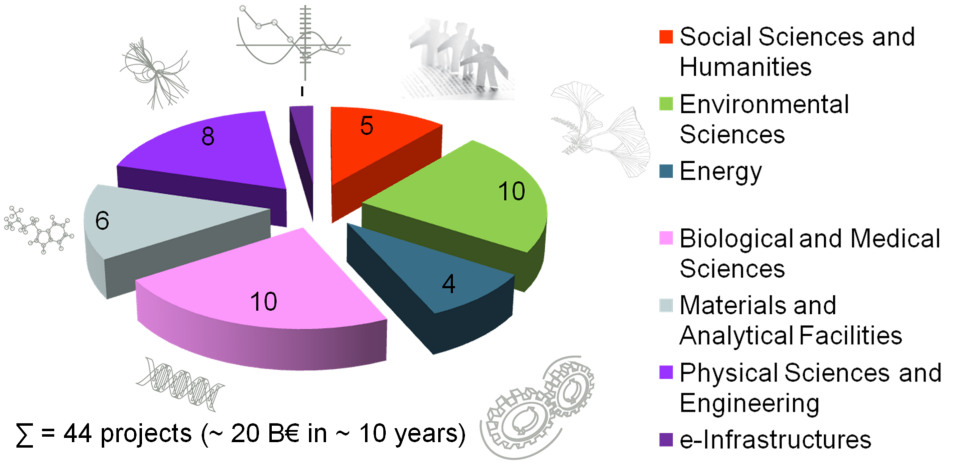

Figure 2. Research areas as listed in the ESFRI Roadmap 2008 and the number of different ESFRI projects within them. We observe that environmental sciences as well as biological and medical sciences have the most projects.

In fact, it is worth mentioning that the ESFRI roadmap update from 2006 to 2008 brought 10 more projects on the list, while the current 44 projects are the only ones that have been selected out of 204 proposals. This in turn means that we can potentially expect a growth of the list in the next coming years to organize even more communities while also many of them are expected to take advantage of the existing computing-based e-infrastructures such as DEISA or PRACE. This in turn leads to a first thought that these 44 current projects represent a hugh set of end-users that already aligned themselves towards one community vision and thus it seem that they can be supported easier than the single researcher many infrastructures have to serve today.

But a more closer investigation of some ESFRI projects reveals the contrary. Furthermore, an open question is too which extend they require computing services as provided today in the e-infrastructures at all. In other words, the ESFRI list of projects envisage their so-called *research infrastructures (RIs)* different from the existing e-infrastructure services landscape (where computation seems to be the most important resource). Their RIs are much more scientifically-driven consisting of a wide variety of different applications, tools, codes, and physical devices reaching from ships to antennas and laboratories. And such RIs may or may not use the basic services (i.e. secure compute and data access) provided by the EMI project that potentially become deployed on e-infrastructures in the next years.

Even for those projects that use the EMI basic services as shown in Figure 3 their is still an open question in many cases if they can use their applications directly with them. Hence, their is a gap between the services provided today and the ESFRI project tools, applications and codes that are intended to be used by e-scientists on a daily basis.

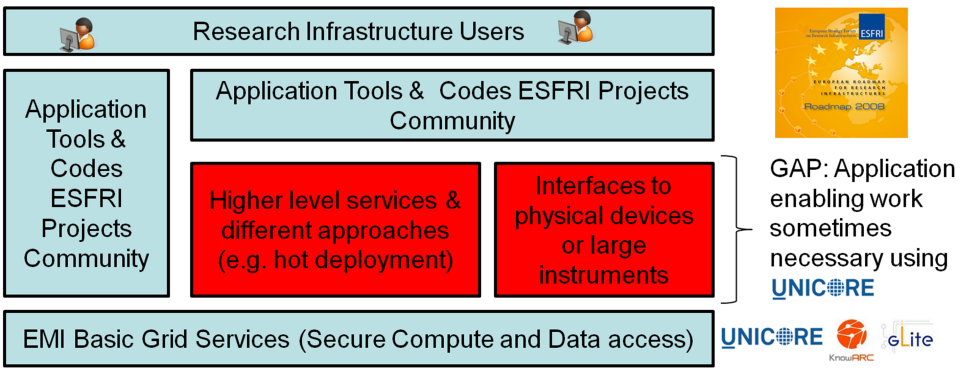

Figure 3. Not all of the ESFRI projects can use the computational-based e-infrastructures like DEISA, PRACE or EGI in a straighforward manner. Potentially many of these projects need application enabling and the UNICORE technology provides an excellent basis to fill this gap with higher level services or interfaces to physical devices.

While this gap seems to be a problem it provides the opportunity for UNICORE to be a basis for higher level services based on the EMI basic services as shown in Figure 3. We refer to this as *application enabling* work that not only means using the extensibility of UNICORE with higher level services, but also to host interfaces to physical devices needed to extract/access data from large instruments for their use in e-Infrastructures if needed.

We thus can add to the UNICORE roadmap the ESFRI roadmap since we expect to have users out of this communities in the next decades mainly because in many cases their first operation starts around 2018-2020 with potentially ever-increasing demands.

5 UNICORE and the VPH Roadmap

As we have added the ESFRI roadmap as one building block to the UNICORE roadmap it is also worth considering the roadmap of the *Virtual Physiological Human (VPH)*[14]. The VPH is a methodological and technological farmework that enables collaborative investigations of the human body as a unique complex system. In contrast, today e-Health scientists take already advantage of e-infrastructures to perform computationally-intensive investigations of the human body but they typically only consider each of the constituent parts separately without taking into account the multiple important interactions between them. The VPH vision is part of the larger international Physiome Project that raises the demand in its roadmap for making world-wide e-science infrastructures interoperable so that it becomes suitable for collaborative research in order to tackle the simulations the VPH addresses. Hence, the capability of a single e-infrastructure is by far not enough to satisfy the VPH vision needs and the true interoperation between production e-infrastructures is still not working properly.

One might think that the VPH is only a dream to engage in the understanding into complex diseases like alzheimer or cancer, but there are specific activities that push this vision so that the dream becomes true some day. Most notably, the EC has organized dedicated calls for projects and thus provides specific amounts of fundings to realize the VPH vision. We provide some examples from the initial (Call 2) VPH projects to support the understanding that the complexity of VPH demands that full projects over years tackle only minor parts of it. The euHeart is developing open source codes and multi-scale/multi-physics models of heart electromechanics in clinical cardiac diagnostic. The ARCH project develops clinical decision support tools based on patient-specific predictive modelling of vascular pathologies. Another project called IMPPACT is developing minimally invasive, patient-specific treatment strategies for liver cancer on bioengineering multiscale modelling principles. The list goes one with having 14 different EU projects that all tackle one specific aspect of the VPH vision only as part of one dedicated EU call. We can foresee that future EU calls will be necessary to realize the VPH vision during the next decade since many tools, applications, and codes are still in the prototype phase.

We have collaborated with members of the VPH community to enable the interoperability between TeraGrid, NGS, and DEISA since especially the patient-specific simulations are very computationally intensive thus requiring more than one e-infrastructure. The access to HPC-based resources in DEISA was provided by UNICORE and it is considered by VPH members as an important technology to interface to HPC resources mostly because of its maturity and having over 10 years of experience in this field. In fact, we already worked on frameworks how UNICORE can be used with one of the VPH tools named as the application hosting environment (AHE). We used this UNICORE and AHE-based framework during production runs on DEISA with, for example, the hemelb lattice Boltzmann code that is designed to simulate fluid flow in the sparse topologies of the patient brains[17]. More recently, we also collabored with members of the VPH community in the context of HPC-based biomechanics in medical applications. To conclude, the VPH roadmap as well as its funding opportunities might become important for UNICORE as well with a potentially higher direct participation in projects than before.

6 Other e-Research Endeavours

The *Cloud computing approach* has entered the e-Research arena while still many experts in distributed computing rather believe that this is clearly oriented towards mainstream commercial users than supporting traditionally smaller scientific communities. Also, many of the concepts of Cloud computing face the same challenges as within Grid computing and e-science. Hence, it still remains distributed computing that tend to create a little bit more dynamic infrastructures while we already experienced that even creating and maintaining static infrastructures is challenging. Questions thus remain whether the Cloud computing approach is really applicable to the e-Research arena in general and the support of e-science environments in particular. Surely, UNICORE, as being an important technology in distributed computing, can also play a major role in Cloud computing using virtualization techniques and such like in the near future. It is thus worth of adding the Cloud computing approach to the UNICORE roadmap, but only as a minor element.

Another interesting new emerging field is *Green IT* that also found its way into the focussed distributed computing arena. In fact, we are already part of the Fit4Green project that is not directly related to UNICORE, but that offers insights into projects that are driven by environmental contraints rather than purely user needs. Also, the shape in computing might shift since power consumption might become soon the boundary in many infrastructures rather than the technology. Even infrastructures face a change, while desktop Grid infrastructures (e.g. EDGI) are currently one of the most promising sustainable infrastrutures where end-users just provide their cycles, GreenIT concepts in future might lead to the fact that end-users are not able to share their cycles anymore because of economic considerations. The role of UNICORE in this emerging e-Research endeavours remains unclear, but it is still worth mentioning GreenIT aspects on the roadmap since it highly potentially becomes more important in the med-term future.

Another building block of the future might be the large *'fusion community'* driven by the European Fusion Development Agreement (EFDA). The idea of fusion are future energy-generating power-plants using the same energy source as for the sun and other stars. There is research in this area since more than 50 years and still not everything is understood since it turns out that such a power plant is a very difficult problem. We have collaborated with the EUFORIA project in terms of using DEISA and EGEE in an interopable manner with the BIT1 and HELENA fusion codes[15]. We thus can state that UNICORE is able to satisfy the demand for many members of the fusion community in terms of HPC access, but it still remains unclear whether all the efforts will lead to success in the fusion community. Nevertheless, when fusion science will go on for the next decades UNICORE might become the technology of choice of using HPC resources most effectively with their own Kepler-based workflow tools and applications.

Finally, *e-Business* where Service Level Agreements (SLAs) play a crucial role are on the border of e-research but interesting since also UNICORE might be considered as base technology for commercial products because of its very open license. In addition, many European projects requires an exploitation plan of how results might become a commercial product and thus it is another building block of the UNICORE roadmap.

7 Summary and Conclusions

e-Science is accepted along the traditional scientific paradigms (theoretic models and practical experiments) and UNICORE is used in DEISA beyond the typical computational science usage based on SSH. We have explored the potentials of using UNICORE in e-science in general and in conjunction with e-infrastructures and virtual research communities. The aforementioned aspects are summarized in the top left within the roadmap shown in Figure 4 where upcoming EU calls bear a lot of potentials for UNICORE.

Due to the infrastructure and resource complexity middleware technologies like UNICORE will become more important in the next decade. Even more important is collaboration and therefore we described the benefits of being part of a technology alliance like the EMI. Benefits for UNICORE-based infrastructures like DEISA or PRACE are access to LHC data and the benefit for UNICORE might be a larger amount of end-user by being also deployed on several EGI sites through EMI. Here, one emerging deployment situation can be that large-scale HPC systems will be accessible in PRACE via UNICORE while the currently low- to medium-scale HPC resources might also become part of EGI over the next years. To conclude, the usage of UNICORE might increase in unprecedented ways and the aforementioned aspects are illustrated in the roadmap at the bottom starting from the left.

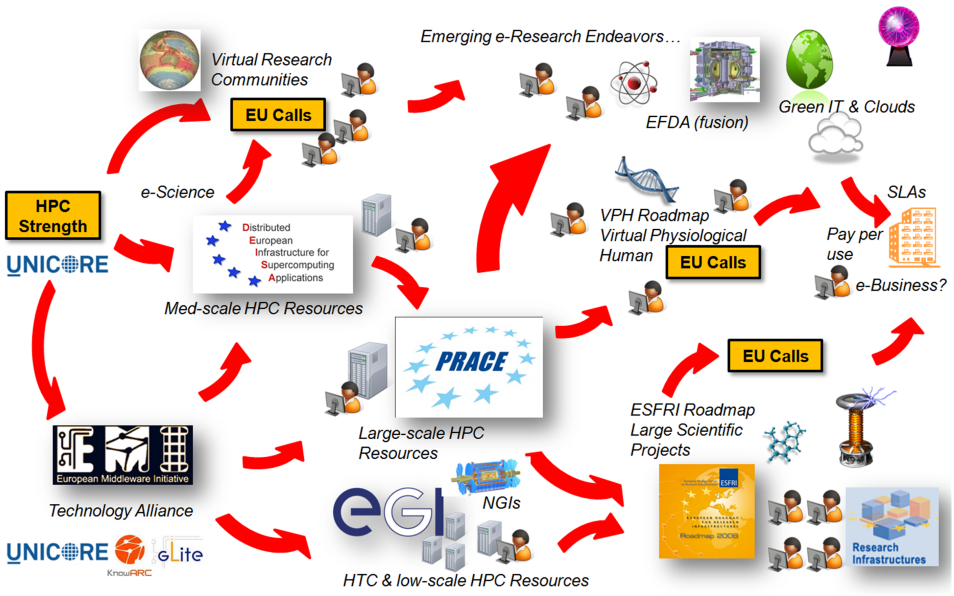

Figure 4. One potential roadmap for the UNICORE technology in the next decade(s) highlighting its key role that remains the provisioning of HPC-driven services in distributed computing infrastructures.

The findings of Section 4 are summarizes in the bottom right standing for the funding opportunities that arise from the ESFRI roadmap and its large instruments. Similiar potentials for funding in the context of the VPH roadmap have been identified in Section 5 and are illustrated right above the ESFRI roadmap. Other rather vague e-research endeavours are depicted in the top right and center right covering Cloud computing, GreenIT, fusion science (e.g. EFDA) as well as e-Business potentially based on SLAs.

To conclude, the potentials for UNICORE and its future are high with a lot of opportunities in each of the aforementioned roadmap segments. One of the rather unusual ESFRI projects is PRACE that serves the needs of other ESFRI projects in terms of HPC and in this project UNICORE is already part of the technologies considerd for production use. Application enabling work is necessary for many scientific approaches and the EMI project provides basic set of Grid services, but this might be not enough. The UNICORE environment provides a perfect basis for higher level services and implementations that interface with large instruments that often only expose legacy interfaces. We have seen that ESFRI and VPH projects require several specialized Web services for e-Research. Furthermore, access to large-scale instruments and emerging data governance demands need to be satisfied as well in an unprecedented scale.

In many of the projects, initiatives, and endeavours surveyed, we speak of demands for decades to come. ESFRI and VPH are strategic roadmaps with objectives for decades and their preparation, construction, and operation project phases will last for decades still being in early phases today. Hence, many new and emerging UNICORE user communities might evolve while the existing user basis gets extended through a wider deployment of UNICORE in different infrastructures such as EGI. In this sense, the era of Grids does not "end" and thus the era of Grids is just about to 'begin'. Maybe under a "new fancy cover name" (e.g. like Cloud computing) while paradigms of distributed computing remain.

UNICORE is well established in Europe and other regions and might become even relevant for e-Business some day. In many cases the key role of UNICORE will remain its strength in HPC and new concepts and technologies will not appear over night. Nevertheless, the UNICORE roadmap also provides potentials for evolutions in different directions being able to serve the needs of a wider ever increasing user basis. One rather general evolution path is that end-user must have an easier access to resources and infrastructures while science gateways as deployed in TeraGrid might be the first initial step, but still provide a lot of barriers for end-users or challenges for resource owners. Another general evoluion path is that the majority of users will have to deal with much more data in the future and thus new methods in working within the data deluge must be explored. Finally, *the only constant we have is continuous change, it is e-Research with UNICORE.*

Acknowledgements

The work presented here is based on the work of many talented colleagues that all share their expertise and vision in the greater UNICORE community. The author thus emphasizes on the fact that the aim of this paper is not to expose personal findings, but to rather summarize emerging potentials for this community and put them in a sequential order.

References

1. M. Riedel et al. *Classification of Different Approaches for e-Science Applications in Next Generation Computing Infrastructures*, Proc. 4th IEEE Conference on e-Science, USA , 2008, .
2. I. Foster et al. *The Grid: Blueprint for a New Computing Infrastructure*, ISBN 978-1558604759, Elsevier , 1998, .
3. J. Taylor, *Initial e-Science Definition*, `http://www.lesc.ic.ac.uk/admin/escience.html`
4. S. Vaerttoe et al. *Advancing Science in Europe*, ISBN 978-1-58603-796-3 , 2008, .
5. J. Taylor et al. *Workflows for e-Science*, ISBN 978-1-84628-519-6, Springer , 2007, .
6. V. Kasam et al. *WISDOM-II: Screening against mulitple targets implicated in malaria using computational grid infrastructures*, Malaria Journal, doi:10.1186/1475-2875-8-88 , 2009, .
7. I. Foster et al. *OGSA Basic Execution Service Version 1.0*, OGF Grid Final Document No. 108 , 2007, .
8. K. Glinos, *The changing world of distributed computing and the role and contributions of the e-Infrastructure programme in this new environment*, `http://www.ogf.org/gf/event_schedule/index.php?id=2001`
9. H. Gardner et al. *Design Patterns for e-Science*, ISBN 978-3-540-68088-8, Springer , 2007, .
10. *Chemomentum EU Project*, `http://www.chemomentum.org`
11. A. Anjomshoaa et al. *Job Submission Description Language Specification, Version 1.0*, OGF Grid Final Document No. 136 , 2008, .
12. *European Strategy Forum on Research Infrastructures (ESFRI)*, `http://ec.europa.eu/research/esfri`
13. A. Sim et al. *The Storage Resource Manager Interface Specification Version 2.2*, OGF Grid Final Document No. 129 , 2008, .
14. *Seeding the europhysiome: A roadmap to the virtual physiological human (vph)*, `http://www.europhysiome.org/roadmap`
15. M. Memon and M.Riedel et al. *Lessons Learned From Jointly Using HTC- and HPC-driven e-Science Infrastructures in Fusion Science*, Proc. Int. Conference on Information and emerging technologies (ICIET), Pakistan , 2010, .
16. *OpenMolGrid EU Project*, `http://www.openmolgrid.org`
17. M. Riedel et al. *Exploring the Potential of Using Multiple e-Science Infrastructures with Emerging Open Standards-based e-Health Research Tools*, Proc. 10th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid 2010), Australia , 2010, .
18. *EMI EU Project*, `http://www.eu-emi.eu`

Experiences with UNICORE 6 in Production

**Michael Rambadt, Sandra Bergmann, Rebecca Breu,
Nadine Lamla, and Mathilde Romberg**

Jülich Supercomputing Centre,
Forschungszentrum Jülich GmbH
E-mail: {m.rambadt, s.bergmann, r.breu, n.lamla, m.romberg}@fz-juelich.de

Grid middleware is an essential offer to the scientists to hide the complexity and the heterogeneity of the underlying systems and architectures. Within the last years the leading Grid middlewares have made an impressive progress from an early experimental state to a well-established and—especially in the scientific community—widely accepted software for production use. UNICORE 6 is one example of such a success story. Like its predecessor UNICORE 5, UNICORE 6 has been established in various international computing centres and important infrastructures such as the European DEISA and the German D-Grid project. Both projects allow the scientists to prepare and submit their jobs and workflows seamlessly and securely to a wide variety of heterogeneous distributed computing resources and data storages. Due to this variety and heterogeneity a UNICORE installation will get complex fast. This paper demonstrates the experience with UNICORE 6 in production gained by the UNICORE 6 support and operation team from Jülich Supercomputing Centre (JSC).

1 Introduction

The UNICORE 6 grid middleware is run and supported in several computing infrastructures including those maintained by DEISA[a] and D-Grid. Experience from these infrastructures shows that there is not only one canonical way to install UNICORE, but that UNICORE has to be adapted to the setting of each computing centre as well as to the requirements of the user communities or virtual organisations. The integration of site administration with respect to user ID management, system monitoring and security policies is essential, as are the tools already established for performing these tasks. Access to applications and execution environments is important to user communities as well as customised access control and interoperability. As a result, tools for and extensions to UNICORE have been developed over time to support site administrators and users in efficiently performing their tasks.

Generally, a production environment requires some consideration before UNICORE or any other middleware can be installed:

- The general infrastructure layout has to be taken into account. It has to be decided which machines will be used to run the site's UNICORE services and whether central services are to be offered or existing services are to be used.
- Other aspects are the interactions between the UNICORE/X service, the target system interface (TSI) and the target system itself. The target system's hardware and operating system, its resource management system, and the installed applications determine the selection and configuration of these UNICORE services.

[a] DEISA started in 2004 as a European project funded by the FP6 programme of the European Union.

- The integration of middleware services into a production site requires the synchronisation of the site's mechanism for user account management.
- The XUUDB (UNICORE user database) management to provide proper authorization and access control.
- Monitoring of all servers and services: grid middleware services installed on distributed systems lead to a complexity which needs automatic procedures for monitoring the health status of the services. This ensures the availability of the services offered by an infrastructure.

This paper focuses on the administrator's experience gained by running UNICORE 6 in production within large Grid infrastructures. The user's experience has already been addressed in M. Rambadt et al.[4]

This paper is structured in the following way: Section 2 gives an example of a complex computing infrastructure to provide an idea as to where the experience is gained from. The tasks to be fulfilled when managing such an infrastructure and its UNICORE services are covered in Section 3. It describes in detail the interfaces between middleware and site administration together with the tools and mechanisms used. A summary in Section 4 concludes the paper.

2 Motivation

The DEISA (Distributed European Infrastructure for Supercomputer Applications) Grid infrastructure is taken here as an example to highlight the components and services, which compose the system, and the complexity of such an infrastructure. Figure 1 visualises the infrastructure. The basic UNICORE 6 components used have been described in The UNICORE 6 Architecture.[5]

Figure 1 shows that UNICORE 6 is available on every partner site's supercomputer. The UNICORE administrators agreed on setting up one central registry at JSC and one backup central registry at CINECA, Italy. Each site has one gateway and both a UNICORE/X and a TSI (Target System Interface) for each target system. All supercomputers are connected via a high performance network. GPFS[6]-mounted file systems guarantee the users' access to all their files from all sites.

Figure 1 gives an impression of how complex a UNICORE installation gets even with only a few (in the DEISA case eleven) connected sites. When setting up UNICORE, the autonomy of each site has to be respected: even if the installed components are almost the same at each site, there are a couple of configuration differences due to different site and user requirements. Each local UNICORE administrator has to make sure that all these different settings are considered in the site's individual UNICORE environment:

- How many machines do I need to run UNICORE?
- Do I need redundant servers in case of any problems?
- How many users are running how many jobs?
- How complex are the jobs?

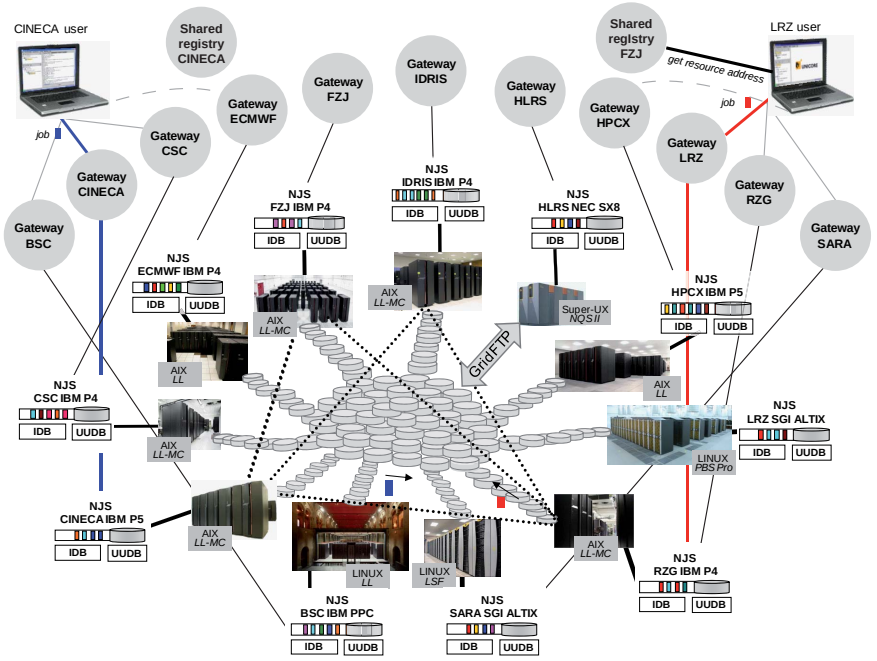

Figure 1. DEISA UNICORE 6 infrastructure[2]

- What kind of jobs do the users submit (workflows or single tasks)?

The decision as to the number of machines necessary is mainly a cost issue. All components can be installed on one machine. This would be the cheapest solution but is also always a single point of failure. The machine could run into memory problems, especially if many jobs are submitted at the same time. Our experiences show that a logical division between the entry layer into the UNICORE Grid (Gateway and Registry) and the application layer (UNICORE/X, XUUDB and workflow services) helps keeping the services stable and reliable.

It is convenient to have an exact copy of the UNICORE installation on separate machines as redundant servers, so that, in case of severe hardware problems, the production can continue with as little consequence to the running as possible.

The hardware requirements strongly depend on the UNICORE usage scenario. The expected number of users and the kind of submitted jobs influence the choice of hardware components. Ample amount of virtual memory is essential to run the UNICORE servers. Our experiences show that the UNICORE/X, which is described in the next section, requires the most memory. Each UNICORE/X should have at least 512 MB to run properly. Otherwise the Java virtual machine easily runs out of memory, which then leads to a crash of the UNICORE/X. The more complex the jobs are the more virtual memory should be available.

UNICORE does not have special requirements as to the installed operating system or software components, except for a SUN Java runtime environment. Of course the

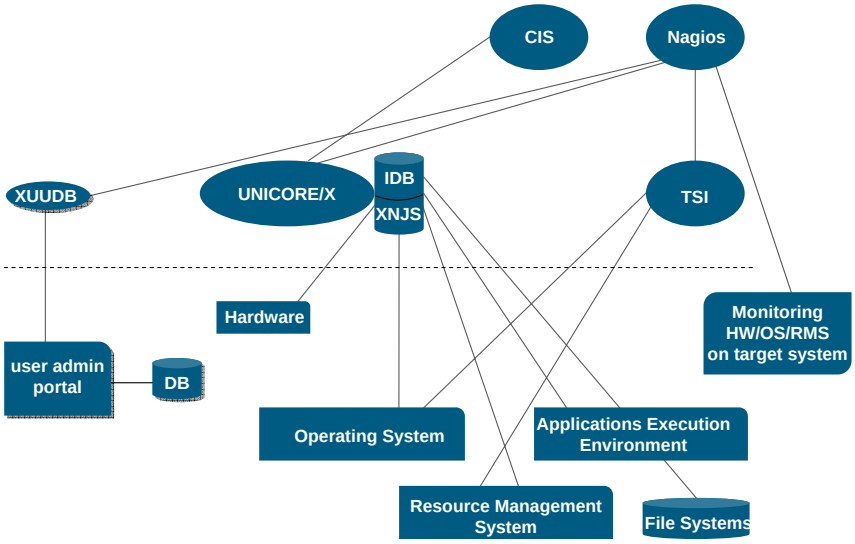

Figure 2. Interaction between UNICORE and non-UNICORE services

UNICORE administrators have to take care about constantly keeping everything up to date by installing all available patches. The users require a stable and especially secure system.

The experience gained is mostly related to the interfaces between the UNICORE services and components and the hardware and software representing a computing site. Figure 2 shows the involved players, which on the UNICORE side are the core services UNICORE/X, XUUDB, TSI, and the monitoring layer. The following sections will elaborate on Figure 2 in detail.

3 Managing UNICORE Production Environments

A couple of UNICORE services require the administrator's interaction and modification to work properly. Essential prerequisites need to be met in order to run the components, such as specific hardware and software. Before starting the UNICORE installation, some general questions have to be answered, as shown in the following subsections.

3.1 UNICORE/X

The UNICORE/X is the core component of each UNICORE installation, which represents the interface from the abstraction layer to the concrete target system's job description. Setting up the UNICORE/X, the administrator has to consider two components in particular: the Network Job Supervisor (XNJS) which handles the job execution and file access, and the Incarnation Database (IDB), which contains rules for translating abstract job descriptions to executable scripts. For the configuration of both components the target system's hardware is relevant. It is important for the user to know the underlying architecture, which

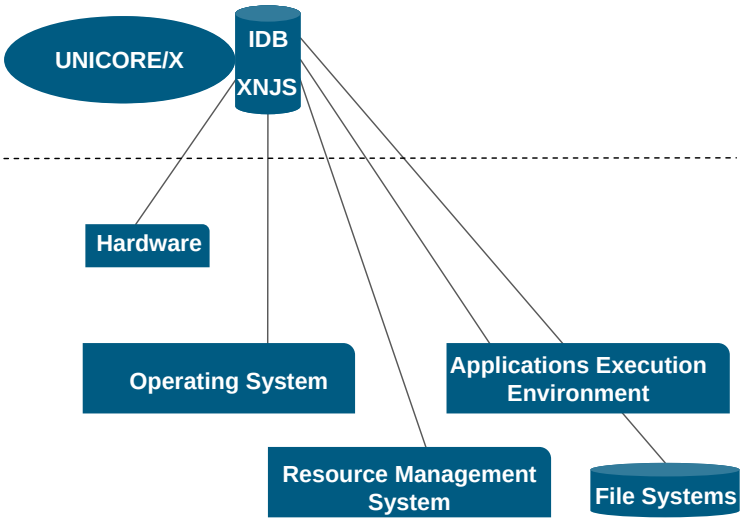

Figure 3. UNICORE/X: Non-UNICORE dependencies to be configured

has a huge impact on the behaviour of the running jobs. Furthermore the operation system provides valuable information for the job creation. All paths to the standard operation system commands are defined in the XNJS.

The IDB is the place to configure the interface with the target system's resource management system. The available supercomputer resources, like nodes, processors, CPU time, CPU architecture, etc., and the respective limits have to be specified.

In addition to these general target system settings, the IDB also stores the settings for the UNICORE Application Executions Environment (AEE)[7]. The AEE provides a comfortable interface for the users to easily configure a supercomputer application; the users can configure this environment individually for their jobs. To enable this feature for the users, the UNICORE administrator has to set up the different environments for applications in the IDB. These might contain input parameters, special predefined variables and pre- and post commands. The UNICORE administrator has to know about the available applications and what parameters are necessary to access these. In case that the UNICORE administrator is not the super computer administrator, there has to be the respective interaction between both.

Finally the file systems, which are available for the user on the remote machine, have to be configured in the IDB. This again is an individual process. The file systems might be either temporary or permanent file systems, both local or distributed ones. DEISA, for example, uses global GPFS file systems.

Figure 3 shows all the dependencies, which the UNICORE administrator has to follow from the UNICORE/X point of view.

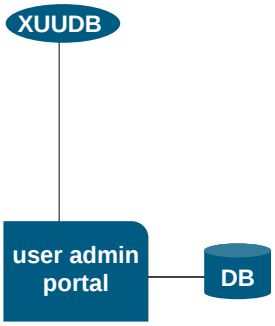

Figure 4. XUUDB: Non-UNICORE dependencies to be configured

3.2 XUUDB

Supercomputer centres provide the computing time on their machines to several projects with a significant number of users each. To handle this amount of users, most centres have automatic user administration procedures, with the user records being stored in arbitrary database systems. In DEISA the user information is stored in distributed databases that are addressed with the LDAP protocol. All these users are potential UNICORE users, and their certificates (or the certificate's distinguished name) and the login names have to be added to the XUUDB.

Up to several hundred users have to be maintained in the XUUDB at the same time for various target systems. Doing this manually is expensive and always a source of errors. Thus scripts have been developed, which automate the XUUDB update process and provide for instance an LDAP interface as shown in Figure 4. At JSC such a script checks all local databases for new users once every night and updates the XUUDB accordingly.

3.3 TSI

Naturally, the TSI is the UNICORE component with the most interaction with non-UNICORE services. On the one hand, the operating system details have to be considered again. On the other hand, the TSI maps the abstract instructions, which are passed on from the UNICORE/X, to concrete batch system settings. The following batch system commands need to be configured: submit, abort, cancel, hold, resume and the commands to receive the batch queue and job status. All the resources, which the user needs to execute her job, are mapped to the respective batch commands. The TSI is the place to make all the batch system logic available for UNICORE; therefore it has to be extendable flexibly.

3.4 Monitoring Layer

The more sites are involved, the more complex a UNICORE infrastructure will get. Therefore it is necessary to establish automatic monitoring procedures like those shown in Figure 6.

The Common Information Service (CIS)[8] provides a general and unified overview of the up-and-running supercomputers, like queue status, the amount of running user jobs,

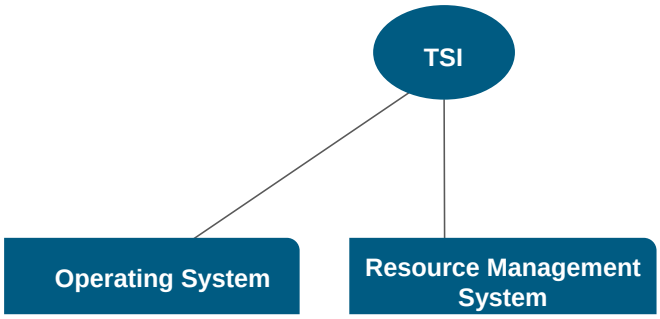

Figure 5. TSI: Non-UNICORE dependencies to be configured

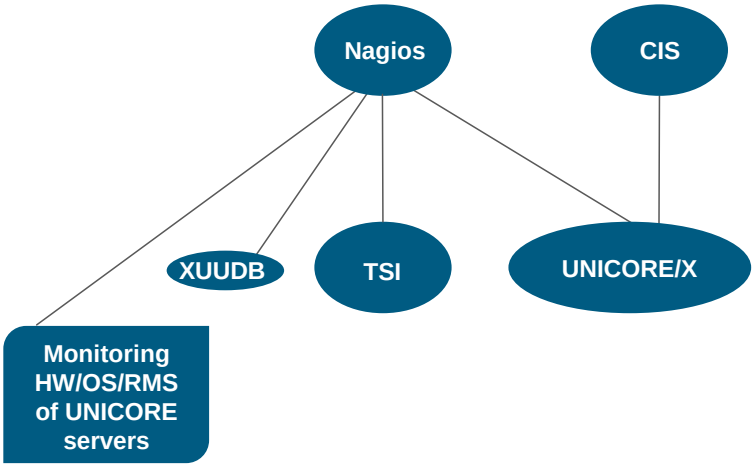

Figure 6. Monitoring Layer dependencies to UNICORE

maintenance information, etc. The CIS interacts with the UNICORE/X, which forwards the CIS requests to the target system.

In a production environment it is also essential to have monitoring procedures which report any cases of failure of the UNICORE components. If something is not working as expected, the administrators have to be informed immediately to initiate further action. Therefore Simon 6[9] has been developed to monitor the health status of each UNICORE service. Simon 6 provides a lightweight monitor by submitting real UNICORE jobs. If such a job fails, the error code is filtered out by Simon 6 and interpreted respectively.

Simon 6 is designed to prepare the results for established monitoring frameworks, like INCA,[10] and Nagios.[11] Both frameworks are widely used in many supercomputing centres and interconnections, like DEISA. Nagios and INCA are extendable in a flexible way, and a couple of additional reporters are available by default already. Simon 6 provides both Nagios and INCA reporters.

To estimate the UNICORE health status, it is not enough to check the UNICORE services themselves, but the general system status has to be considered, too. Here, the system is not the supercomputer, but the machines on which the UNICORE services are installed. Important parameters are, for instance, the general hardware status, the number of processes, and the current load of the system. The more information is available, the better the monitoring will be.

An interesting aspect for administrators is how frequent the monitoring tests should be performed. Following the approach that real jobs are submitted to check the health status of UNICORE, the administrator has to take care that the monitoring is not too costly. Even if the submitted test jobs only require minimal resources and minimal execution time on the target system, they block resources for the time running. As a consequence, the supercomputer administrators generally argue for executing the UNICORE test suite as infrequent as possible. But on the other hand the Grid middleware administrators want to have the smallest intervals possible between all the tests to get a coherent view on the Grid middleware health status. Based on intense discussions with site and system administrators, within DEISA a five minutes interval turned out to be a reasonable compromise. If a failure is reported the administrators can react for the short term.

4 Summary

During many years of experience in administrating UNICORE production environments, we have come to the conclusion that there is no standard UNICORE installation. Different user and site requirements have to be considered for each new site. Running UNICORE production environments is complex, but due to its flexibility UNICORE can be easily adopted to each site condition. This flexibility allows the proper interaction between UNICORE core services and non-UNICORE services.

To fully integrate the UNICORE infrastructure into the local site management procedures, the development of additional administrative tools is essential. Therefore, at JSC we have developed tools to monitor the UNICORE services and to update the XUUDB with the user records from LDAP, automatically. Our intention is to automate nearly everything to minimise the risk of individual administrator failures. But even with all this automation, a minimum of manual interaction is not to be avoided, for instance the mapping of batch system commands in the TSI or the individual configuration of the IDB.

In addition to all the technical interactions, the social aspect has to be respected as well: UNICORE provides the layer between the user and the supercomputer, so the communication with both the scientists and the supercomputer administrators is essential for successful production.

References

1. A. Streit, S. Bergmann, R. Breu, J. Daivandy, B. Demuth, A. Giesler, B. Hagemeier, S. Holl, V. Huber, D. Mallmann, A.S. Memon, M.S. Memon, R. Menday, M. Rambadt, M. Riedel, M. Romberg, B. Schuller, and Th. Lippert, *UNICORE 6—A European Grid Technology, High Speed and Large Scale Scientific Computing*, Editors: L. Grandinetti, G. Joubert, W. Gentzsch; Amsterdam, IOS Press, 2009, Advances in Parallel Computing Vol. 18., ISBN 978-1-60750-073-5, pp. 157–173.

2. *DEISA Infrastructure and Resources*.
   http://www.deisa.eu/services/infrastructure
3. *DEISA Infrastructure and Resources*.
   http://www.d-grid.de/index.php?id=539&L=1
4. M. Rambadt, R. Breu, L. Clementi, T. Fieseler, A. Giesler, W. Gürich, P. Malfetti, R. Menday, J. Reetz, and A. Streit, *Experiences with Using UNICORE in Production Grid Infrastructures DEISA and D-Grid*, International Symposium on Grid Computing (ISGC 2007), Springer, ISBN 978-0-387-78416-8, pp. 121–131.
5. *The UNICORE 6 Architecture*.
   http://www.unicore.eu/unicore/architecture.php
6. *GPFS*. http://www-03.ibm.com/systems/software/gpfs/
7. B. Demuth, S. Holl, and B. Schuller, *Extended Execution Support for Scientific Applications in Grid Environments*, UNICORE Summit 2010, to appear.
8. A.S. Memon, M.S. Memon, Ph. Wieder, and B. Schuller, *CIS: An Information Service based on the Common Information Model*, Proceedings of 3rd IEEE International Conference on e-Science and Grid Computing, Bangalore, India, December, 2007, IEEE Computer Society, ISBN 0-7695-3064-8, pp. 465–472.
9. M. Rambadt and M. Romberg, *Simon 6—Monitoring UNICORE 6 Resources*, CGW'09 Proceedings, Editors: Marian Bubak, Michal Turala, Kazimierz Wiatr; ACC CYFRONET AGH, Krakow, ISBN 978-83-61433-01-9, February 2010, pp. 134–142.
10. *INCA*. http://INCA.sdsc.edu
11. *Nagios*. http://www.nagios.org

UNICORE Runtime Administration

Milad Jason Daivandy[1], **Bernd Schuller**[1], **and Bastian Demuth**[1]

Jülich Supercomputing Centre,
Research Centre Jülich, 52425 Jülich, Germany
E-mail: {j.daivandy, b.schuller, b.demuth}@fz-juelich.de

Given its characteristics (heterogeneity, operation and maintenance real-time demands), a Grid infrastructure raises specific administrative requirements: (1) provide an administrator with data broad enough to convey an adequate overview, but also fine enough to zoom in on critical areas; (2) give him means to act on the information thus delivered, preferably at runtime, to avoid operational interruption. This paper shows how this has been achieved for the UNICORE Grid middleware by extending the server side with capabilities for dynamic service deployment, runtime reconfiguration and metrics for monitoring purposes. To leverage the additional functionalities and information thus added, the client side has been complemented with a graphical user interface representing the new features in a cohesive manner to maximize their usability. Subsequent tests have shown that the server-side extensions result in a negligible performance impact on a running UNICORE installation. Proposals for improvement have been compiled for follow-up work.

1 Introduction

Used in a decentralised distributed system, a Grid middleware would greatly benefit from means allowing for remote administration at runtime. The underlying idea is that enabling maintenance of a system without having to shut it down increases its availability, itself being a crucial aspect of every kind of middleware. This paper describes how to extend the UNICORE Grid middleware with runtime administrative capabilities with minimal changes to UNICORE core concepts so as to cut development overhead, guarantee downward-compatibility and avoid operational performance drops. Before outlining design and implementation decisions for the developed solution, some basic concepts are introduced:

1. *dynamic (un)deployment* denotes the capability to introduce new code to (remove existing code from) a software system at runtime
2. a *metric* is considered a software entity deployed at strategic points within a software system to gather data and compute from these results according to predefined criteria (e.g. performance, reliability etc.)
3. *runtime reconfiguration* comprises all means allowing for configuring a software system without having to initiate a restart

Regarding this, different concerns come into play, each one individually independent, but collectively leading to the following list of concrete tasks that have been performed in the scope of this work:

1. provide metrics at strategic points within UNICORE for representing system health and performance

2. make UNICORE services (un)deployable at runtime
3. make UNICORE configuration modifiable at runtime and actions executable based on configuration changes
4. deliver a graphical user interface (GUI) to facilitate access to and control of the aforementioned features

Chapter 2 gives a rough overview of the UNICORE architecture, outlines general design decisions and covers crucial implementation aspects for the aimed-for extensions, including a detailed description of the powerful, yet easy-to-use GUI for leveraging the runtime administration extensions. Chapter 3 deals with the outcomes of putting the solution to the test, whereas chapter 4 gives a short overview of related work and as to how our solution contrasts with it. Finally, chapter 5 concludes with an outlook on potential future work on this topic.

2 Design and implementation

UNICORE is separated into three layers (cf. fig. 1): as the top-most layer, the *client layer* provides end-users with different ways to access UNICORE. Authenticated and authorised end-users use services of the *service layer* (implemented as Web services) providing, among other things, access to target system resources through job and storage management services. Finally, the *system layer* connects to actual resource management and batch systems and is utilized by the service layer.
The extensions made in the course of this work apply to the service and client layers, respectively.

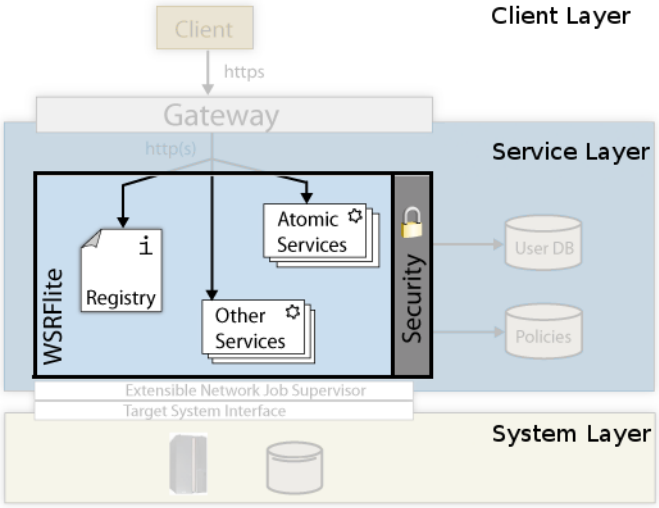

Figure 1. The UNICORE 6 architecture consists of three layers.

2.1 Dynamic service deployment

Adding and removing UNICORE services at runtime without operational interruption would provide an administrator with a powerful maintenance edge towards the traditional way of having to restart the Grid middleware to add new functionality.
To achieve this, UNICORE's underlying Web service framework *WSRFlite* had to be extended to allow for dynamic deployment of Web services. This feature has been implemented by replacing the previously existing static deployment mechanism with *DeploymentManager*, a dedicated dynamic service deployment component built for this solution. This also comprises automatic modification of service-related configuration files.
DeploymentManager was designed to distinguish between two service types, *Core services* and *Plugin services*. The *UNICORE Atomic Services*, a set of Core services, responsible for basic concerns like file transfer, job and storage management, belong to the former service type which can only be (un)deployed after restarting UNICORE. Added functionality in the form of new services, on the other hand, is covered by the Plugin service type. A Plugin service is delivered via a Java archive (JAR) containing the compiled service code and a deployment descriptor containing deployment-specific metadata. Among these is the optional initialisation task covering the potential need for custom service-specific initialisation logic by specifying a task to be run before the service is being deployed. Service deployment itself imposes one critical requirement: service name uniqueness must be maintained. Since there can be concurrent dynamic service deployment requests, service name collisions have to be resolved.
Upon UNICORE startup, each Plugin service archive is tested for integrity by comparing its current checksum to the checksum computed after successful initial dynamic deployment. If both checksums don't match, the Plugin service in question is automatically removed from UNICORE.
Additionally, DeploymentManager features *AutomaticDeployment*, a periodic task checking for new Plugin service JARs copied to a specifiable directory. Thus, a local administrator can easily install new services while UNICORE is running.
To keep UNICORE downwards-compatible from an administrator's point of view, it is installed with dynamic service deployment switched off. Configurability is fine-grained enough to toggle dynamic service deployment, AutomaticDeployment and remote deployment (cf. 2.4).

2.2 Runtime reconfiguration

UNICORE is configured by a set of files containing system properties governing its behaviour in different aspects:

1. Web service configuration: HTTP and application server, authentication, persistence
2. Service access policy: rule-oriented user authorisation
3. UNICORE-specific properties: site names, registry settings, user database, startup logic to be executed etc.

metric	unit	target
call frequency	calls / sec	each UNICORE service
mean processing time	ms	each UNICORE service
minimum processing time	ms	each UNICORE service
maximum processing time	ms	each UNICORE service
current processing time	ms	each UNICORE service
mean wsrf processing time	ms	all UNICORE services

Table 1. Metric types applied to UNICORE 6

Hence, UNICORE's performance and behaviour can be tweaked by controlling these properties. This made a strong case for integrating mechanisms allowing for system property access and modification and acting upon those changes in a predefined manner – both at runtime.
In the course of this work, runtime reconfiguration extensions have been made for the Web service configuration. To enlarge upon its aforementioned characteristics, it is used to set HTTP server parameters, authentication settings, service access properties, and which persistence back-end to use. Each Web service configuration property can be enabled for remote access (read or read-write). To accommodate the eventual need for running code upon property value changes, each property can be assigned a *PropertyListener*. This is another new component that, as the name suggests, listens to each value change performed on the specific property and performs specifiable actions if specifiable prerequisites are met. These can either be simple value changes or transitions from specific old values to specific new ones. Thus, a PropertyListener can act as a state machine, allowing for complex behaviour where necessary.

2.3 Metrics provisioning

Metrics provisioning was realised by developing and using *metriX*, a stand-alone lightweight metric framework written in Java, employable in any Java context. metriX provides an easy way to write custom metrics enabling programmers to focus on task-specific logic. A metric can measure whatever it's programmed to. A few examples, used for this work, can be found in table 1.

The central *MetricRegistry* keeps track of all registered metric instances and is used for metric retrieval. Since metrics can be categorised (e.g. performance, start-up, system etc.), metric look-up can be done for a given set of categories.

The intended metric lifecycle is as follows: use available metric types or create your own metric type(s).

- instrument UNICORE source code: create adequate metric instances at strategic locations (e.g. potential performance bottlenecks) in the source code and push data to each metric instance

each metric instance

- each data push triggers the task-specific processing logic of the respective metric instance and yields a result, formed as a tuple *{name, value, unit, capture time}*
- said result is pushed to metric subscribers. Additionally, the most current result can be directly retrieved from a metric instance.

Since each metric instance usually deals with a lot of subsequent data pushes (cf table 1), there are usually as many results. While the most current result can always be retrieved from a metric instance, previous results aren't stored. Should traceability be required, however, components can be subscribed to each metric instance, allowing for notification of every new result. A persistence component, for example, could be subscribed to all metric instances in the system. Each of these metric instances would notify it of new results, which then could be persisted.

2.4 Administration facade

Bringing the aforementioned server-side extensions together for remote server administration has been realised by creating *AdminService*, a Web service dedicated to administrative tasks. It unifies access to dynamic service deployment, server-side properties and metrics. Remote administrators can use it to (un)deploy Plugin services, retrieve metrics data, inspect service resources, set service resource lifetimes and get/set system properties flagged as remotely accessible. Access to AdminService is governed by the regular service access control mechanism used in UNICORE, thus assuring that only privileged users can use the aforementioned features.

2.5 Cohesive client-side control and data representation

To leverage the disparate information retrievable through AdminService and represent it in a useful and cohesive manner, an appropriate GUI was needed: *AdminDashboard*. It was decided to design and implement AdminDashboard as a plugin for the *UNICORE Rich Client* (URC), an already existing GUI to UNICORE based on the *Eclipse Rich Client Platform*.
AdminDashboard presents itself as a tree, with *AdminGrid* being the root node. Connection to UNICORE sites is achieved by connecting to AdminService instances (cf. 2.4), resulting in AdminService child nodes to be added to AdminGrid.

With future extensions in mind, AdminService nodes expose functionality by distinguishing in three areas of concern, each represented as a child node, providing available actions as context menu entries.
ServerConfigurator is used for configuring server-side settings (cf. 2.2). Currently, Web service settings can be modified through an interactive dialog for reading and writing remotely accessible server properties, potentially triggering predefined responses. This feature has a built-in undo command for the most recently changed settings property.
The aforementioned dialog is modeless (i.e. doesn't block access to background GUI elements) to accommodate the potential need for juxtaposing multiple such dialogs of

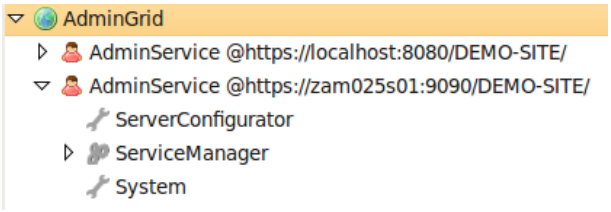

Figure 2. AdminGrid node consisting of AdminService nodes.

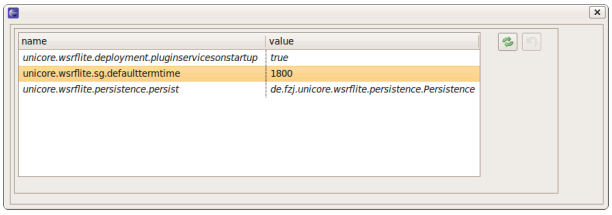

Figure 3. Configure UNICORE 6 server properties.

multiple remotely managed UNICORE servers, e.g. for comparison purposes.

ServiceManager handles all things related to runtime management of services. Each service is displayed as a child node, each of which respectively containing resources as child nodes, if available. Services can be deployed and undeployed, resources can be viewed in detail.

The *System* node offers a *show metrics* action, which pop ups an interactive modeless dialog consisting of two tabs: *Metrics* and *Watched Metrics*.
The Metrics tab is used for listing metrics in a table: one row for each metric instance, the first four columns representing its tuple values (cf. 2.3), the last one stating whether it's currently being watched. Clicking inside it starts and stops watching the selected metric. A dropdown menu serves as a category selector, effecting only metrics of the selected category to be listed. Additionally, clicking on a column header appropriately changes the listing order.
The Watched Metrics tab allows for visualising metric data in time series (one graph per metric), each graph can be singularly reset. Server load is kept at a minimum by polling all watched metrics in a single request. The polling interval is customisable, with a minimum value of 5 seconds.
Additionally, minimum and maximum threshold values can be set per watched metric, resulting in user notification in case of falling below minimum or exceeding maximum threshold values. For this purpose, an administrator can choose among two notification levels: *INFO* (background logging) and *SEVERE* (notification by pop-up dialog).

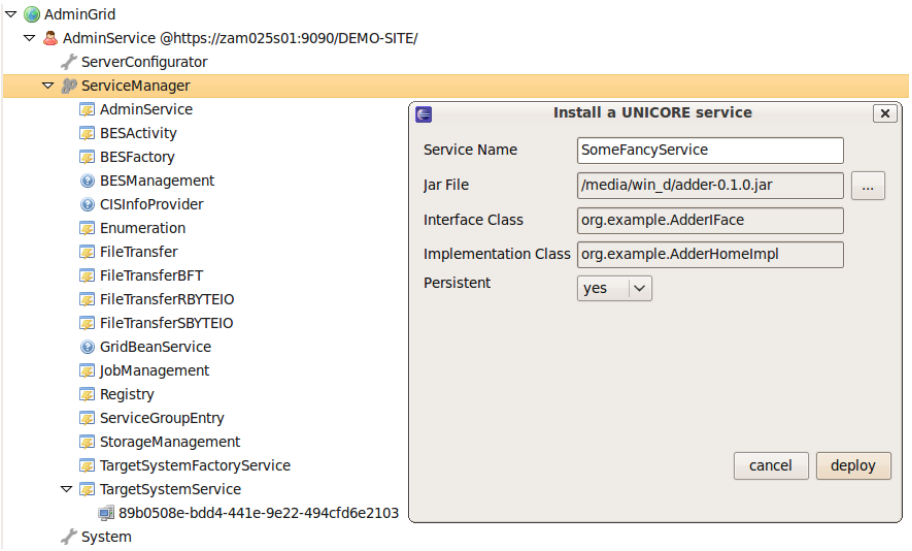

Figure 4. Manage UNICORE 6 services.

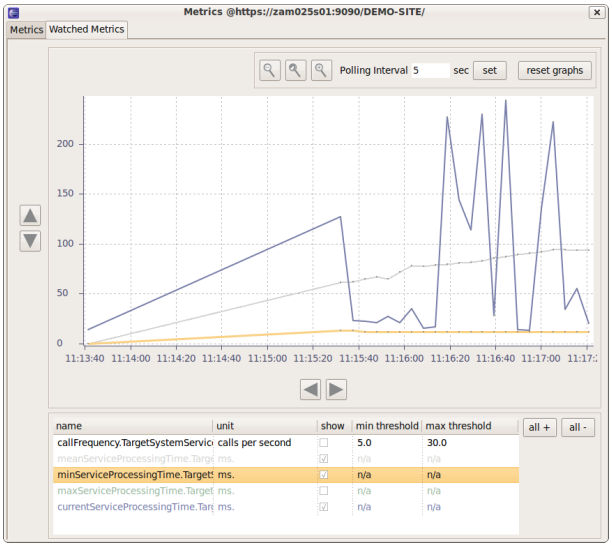

Figure 5. List and watch metrics, assign threshold values and notification levels.

3 Tests

To verify negligible performance impacts caused by the runtime administration extensions, the examination focus was set on the dynamic deployment feature, since the remaining features don't pose much of a demand.

UNICORE was installed and run on a system with the following specs:

- CPU: 2x Intel(R) Xeon(TM) 2.80 GHz CPU
- RAM: 6 GB
- HDD: 2 x 250 GB (RAID 1)
- Operating system: SuSe Linux 11.1

Two criteria have been tested: (1) performance impact of dynamic service deployment on a UNICORE system in production and (2) scalability of dynamic service deployment on an idle UNICORE system.
For both criteria, measurements have been taken by instrumenting UNICORE TargetSystemService and AdminService.deploy() with appropriate metrics (cf. 1): mean processing time, minimum processing time, maximum processing time.

3.1 Performance impact on production system

A reproducible and qualitatively representative reference performance profile was achieved by applying a load generator specifically developed for this purpose. Essentially, it puts stress âĂŞ constant in average âĂŞ by executing 10000 reading operations on *TargetSystemService*, a UNICORE Atomic Service responsible for providing user access to target systems.

TargetSystemService performance was measured both in solo and in parallel with dynamic deployment requests via AdminService; the solo results serve as reference values. Dynamic service deployment requests have been dispatched in different numbers (5, 10, 20) and volume (398 kB, 733 kB, 1400 kB), representing small-, medium, and big-sized service JARs.
Effectively, this setup resulted in 10 test runs: one test run to establish the aforementioned reference values and nine additional ones to determine TargetSystemService production performance when run in parallel to dynamic service deployment requests. UNICORE has been stopped and restored to a clean slate (removal of Plugin services and their persisted data) to allow each test run to operate in comparable fairness.

The test results show that performance impact on TargetSystemService performance is negligible. From a mean processing times point of view, performance impacts are very low, whereas the respective maximum processing times show that performance is compromised by about 25 %. All in all, that decrease is only temporarily, though, just occurring during dynamic service deployment requests. The reason for this performance decrease lies mainly in TargetSystemService and AdminService.deploy() requests competing with each other: considering that Web service calls occur synchronously (request/response pattern) and that calls to AdminService.deploy() require unpacking the supplied Plugin service JARs this is expected behaviour.

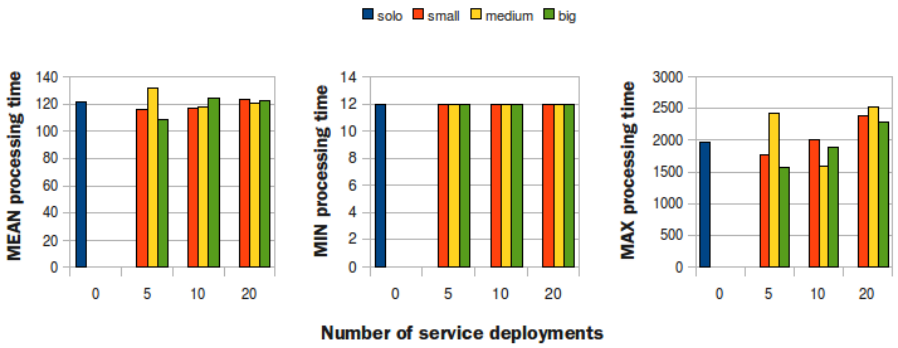

Figure 6. Processing times for TargetSystemService

3.2 Scalability of AdminService.deploy()

Testing for scalability of dynamic service deployment via AdminService has been set up similarly to testing the performance impact on TargetSystemService, albeit without applying the load generator.

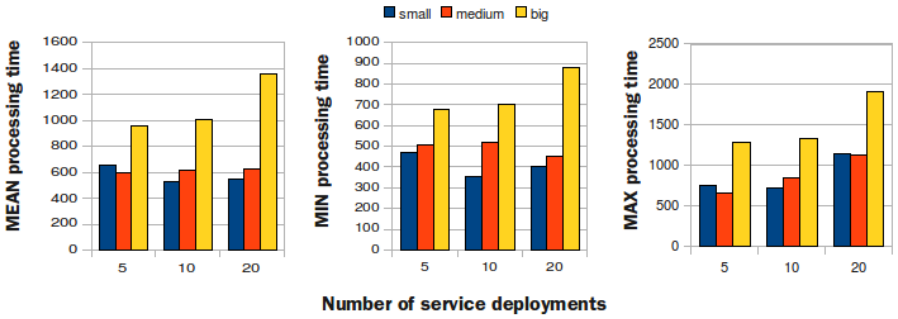

Figure 7. Processing times for AdminService.deploy()

As is evident from fig. 7, dynamic service deployment via AdminService scales well.

4 Related Work

The work presented in this paper combines ideas and solutions from different areas of concern to improve on UNICORE 6 administrability.

DynaGrid[1] extends GT4 (Globus Toolkit 4) with dynamic service deployment capabilities and supports service resource migration. Being independent from GT4, it is essentially a fully-fledged dynamic service deployment framework.
HAND[2] uses a different approach by directly building on GT4 and differentiating in dynamic service deployment on container and service levels. The latter denotes reloading (and thus re-initialising and reconfiguring) the service container after a service has been dynamically deployed, whereas the former doesn't necessitate any changes to the service container after dynamically deploying a service.
Our solution contrasts with the aformentioned ones in terms of provided features and complexity: among others, the scope of this work was to be able to (un)deploy services at runtime instead of building a complete framework for dynamic service deployment.

Concerning dynamic container reconfiguration, HAND has to set all services and resources unavailable to the consumer before initiating a container reconfiguration, whereas our solution allows for real runtime reconfiguration without service interruption, albeit in a more limited fashion (e.g. changing the HTTP port is not possible).

5 Conclusion and outlook

This paper has shown which steps have been taken to extend UNICORE with runtime administration capabilities and put the dynamic service deployment mechanism to the test, thereby proving that performance impact on a UNICORE production system is fairly negligible.

However, Plugin service JAR sizes are limited, since they are attached to each dynamic service deployment request and thus can require a rather big portion of system memory of the UNICORE server depending on the JAR file size. Instead of attaching a Plugin service JAR to a dynamic service deployment request, an alternative way of transferring this data should be offered; data stream based approaches, for instance, are known to perform far better in this regard, since only small chunks of data are handled at a time as opposed to handling the whole data at once. To implement this, the dynamic service deployment process needed to be separated into three phases, for example: (1) notify UNICORE server of dynamic service deployment request, (2) upload Plugin service JAR for this exact request, (3) deploy. That requires altering the dynamic service deployment procedure in such a way as to accommodate the aforementioned asynchronous sequence of events.
This approach could serve as a basis for related concerns. For instance, if a lot of big Plugin services needed to be deployed, steps 1 and 2 could be performed in advance for all of them, in order to issue step 3 to deploy them back-to-back. Apart from deployment timing advantages, this would result in an even smaller performance decrease of a UNICORE production system.
On a different note, the Watched Metrics tab of AdminDashboard could be extended to serve as a fully-fledged metric data visualisation tool: save captured metric values as a capture profile (possibly in different formats, e.g. XML, CSV etc.), restore capture

profiles and compare capture profiles with each other to identify potential performance bottlenecks. Thus, specific capture profiles could be composed to measure and evaluate system performance, e.g. before and after modifications/updates to UNICORE or simply for a day-to-day performance comparison of a UNICORE production system.
Additionally, the extensible nature of AdminService nodes in AdminDashboard could be leveraged by adding further features: one could, for example, extend the ServerConfigurator node with a feature allowing for remote editing of how abstract job definitions are to be mapped onto real executables.
Concerning AdminDashboard in general, it will be vital to encourage UNICORE Grid administrators and operators to use it in order to gather essential feedback for further iterations.

References

1. Eun-Kyu Byun, Jin-Soo Kim, *DynaGrid: A dynamic service deployment and resource migration framework for WSRF-compliant applications*, Parallel Computing **33**, 328–338, 2007.
2. Li Qi, Hai Jin, Ian Foster, Jarek Gawor, *HAND: Highly Available Dynamic Deployment Infrastructure for Globus Toolkit 4*, Parallel, Distributed and Network-Based Processing **15**, 155–162, 2007.

Monitoring of the UNICORE middleware

Piotr Bała[1], **Krzysztof Benedyczak**[1], **and Mariusz Strzelecki**[1]

Faculty of Mathematics and Computer Science
Nicolaus Copernicus University
&
Interdisciplinary Center for Mathematical and Computational Modeling
Warsaw University
E-mail: {bala, golbi, szczeles}@mat.umk.pl

The aim of this paper is to present a system which was created to provide monitoring facilities for the UNICORE grid in the PL-Grid project. The monitoring system is based on the Nagios monitoring framework and consists of numerous plugins. To minimize overhead produced by the monitoring actions, detailed dependency graph was created. The system is deployed in the PL-Grid and is used to detect erroneous situations and to diagnose failure reasons.

This paper describes existing tools that can be used to fill monitoring tasks, covers current achievements and presents future plans for the monitoring system: more fine-grained tests which will allow for a better problem diagnosis.

1 Introduction

Production deployment of any complex distributed system is always a cumbersome task, even if the software quality is very high. Outages can happen due to the numerous reasons. Hardware failures, misconfiguration and network problems are the most common ones. Status monitoring and fast fault detection are therefore necessary to provide high quality of service.

Monitoring of distributed system is not a trivial task. The data has to be gathered from large infrastructure and aggregated. Due to the size of the system, it may consume significant resources on its own, just to provide detailed information on the system status. A good monitoring system has to be scalable and has to provide minimal overhead to the grid infrastructure.

Our work is concentrated on monitoring of the UNICORE middleware[1,2]. The development is deployed in the Polish Grid run by the PL-Grid project[3] but we were committed to the idea of the creation of a truly universal solution which can be used to monitor any UNICORE deployment.

This paper is organized as follows. Section 2 concerns main goals for grid monitoring. Existing UNICORE monitoring tools are outlined in sec. 3. Sections 4 and 5 describe precisely the monitoring infrastructure being developed. The paper is concluded with an overview of the current and future work.

2 Grid monitoring in general

First of all, we will focus on the reasons for grid monitoring. The most important thing is to *ensure reliability* of a whole grid installation. Monitoring system can also *produce*

statistical data about, for example, a mean time of a job submission and show them in an easy to read, graphical form.

The next task of a monitoring system is to *inform users about current grid state*. For instance, user should be warned if resources are not properly working or if installed applications are too old. Another function is an ability to quickly check if each and every service of a newly installed or just updated middleware works properly.

Grid monitoring can be divided into three levels:

1. System-level grid site monitoring (at cluster level): checking free memory, load and other properties on nodes. This is what the well known Ganglia suite[4] does.

2. Functional grid monitoring: discovering a health of a grid by external checking of functions which should be available to the users.

3. Server-side middleware monitoring: checking the state of middleware that is hidden to users but also important for proper work of a middleware. Such tests can reveal problems occurring from time to time (errors spotted by specific users, with a selected applications etc.), problems which decrease grid's performance (job submission failures which are hidden by automatic resubmission) and other.

Our work is concentrated on the two later points. We believe that the first issue is well covered by generic (not grid related) system monitoring tools.

3 State of art in monitoring

There are several universal monitoring frameworks which are typically deployed to monitor IT infrastructure. One of the most popular, open source, general purpose solutions is Nagios, but there are also many other systems like Zenoss[5] or OpenNMS[6]. The monitoring framework usually provides functions to manage probes (invoking on demand and scheduled tests), organize them logically, manage notifications and persist results. Often a set of universal probes is available for a monitoring framework and additional ones can be developed and plugged in.

Considering specific task of UNICORE monitoring we are aware of two tools.

The first is the *Common Information Service* (CIS). CIS is an information service. It collects data from site probes, which is interesting for monitoring. However, as its main aim is to act as an information service there is no option to notify administrators about problems. Also range of monitored facilities is limited and there is no possibility to invoke tests on demand. From user's point of view it is a good tool to check if required site is available, and whether it has not too many waiting jobs.

The second solution - SIMON 6[7] developed at FZJ is a typical monitoring system. SIMON 6 is a standalone solution, implemented in Perl. It can be integrated with INCA[9] as well as Nagios[8].

We decided to develop our own infrastructure independent from SIMON 6 effort due to numerous reasons. The most fundamental one is that the system is not publicly available. However, there are other problems related to flexibility and amount of information produced.

4 The PL-Grid approach to UNICORE monitoring

PL-Grid project[3] runs Polish National Grid Infrastructure composed of the resources of the five largest supercomputing sites. gLite and UNICORE middlewares are being deployed to provide access to the storage and computing facilities. The PL-Grid infrastructure deploys nearly all of UNICORE 6 components and strives to provide high quality of service. Therefore, a scalable monitoring system with the ability to check the status of every service and to inform quickly which UNICORE module fails must be used, what was the purpose of our effort.

As UNICORE monitoring in PL-Grid must be integrated with gLite monitoring, we were constrained to use a common framework. The Nagios system was chosen as a platform because of its earlier adoption for gLite monitoring and a good feedback from the administrators.

The Nagios platform allows for an extremely easy integration of new probes. It is enough to provide executable which returns a specific exit code and standard output in a predefined format. Additional Nagios features provide possibility to define tests dependencies.

Before presenting the concepts underlying our design we would like to underline that in a large system it is not enough to provide information that one functionality is not working properly. As we can observe tens of problems per month (as the PL-Grid infrastructure is far from being complete), it is essential for administrators to get as precise indication of exact failing *component(s)* as possible. For instance sole information that workflow job is failing is not enough; the monitoring system should provide name(s) of component(s) (e.g. a user database) which are the actual problem cause.

The basic building blocks of our system are Nagios probes. We have created (or will create) tests for all externally accessible UNICORE components. Moreover special application probe is provided. This specialized test can be configured to invoke applications exposed via UNICORE and check whether the application itself is working properly.

When a lot of tests need to be carried out, there is a fear that every of them will be executed, which can result in making numerous unnecessary overhead. Therefore, a dependency tree (fig. 1) was developed for all services *to make minimal amount of checks and simultaneously ensure that grid works properly*. In this graph arrows stand for negative dependencies, that is dependencies that block execution of tests if their parents fail. For example, the system will not check UNICORE/X if Registry test or UVOS test failed last time. If the arrows directions are reversed, positive dependencies can be seen — all services are positively dependent on integration tests. If component A is positively dependent on B then we can be sure that if B is working correctly then A too. For instance no service needs to be checked if Workflow test passed last time.

The presented dependencies assume clever and detailed tests of every component. Referring to the above workflow example we can state that other services are working properly when workflows are performed correctly *only if* those workflows really use all components (all target systems, storages etc.). What is more administrators must be able to easily redefine the dependencies, as those presented on the fig. 1 can be invalid in some deployments.

As the described system is quite large (there are different kinds of dependencies, and many probes) we tried to simplify its maintenance and installation. An installation script

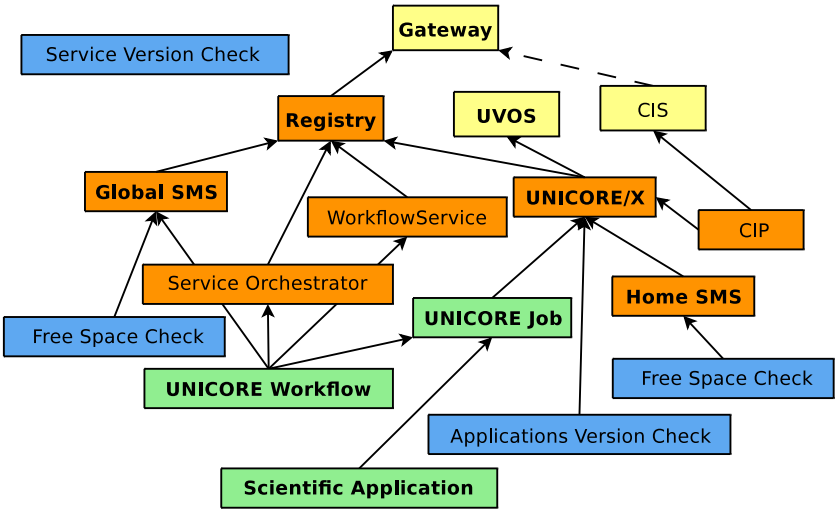

Figure 1. Dependency graph for components tests.

that configures every test was written. Almost all scripts are written in Perl language, which allows them to *run on any operating system*. To get data from services UNICORE Command line Client (UCC) is mostly used — it is the simplest and natural way to check most of the services availability and functionality. However, in some cases other programs or approaches are used, e.g. to check UNICORE Gateway a custom HTTP client was written.

5 Developed tests

As it was mentioned in the above section, separate tests for almost all UNICORE services, including Workflow system and UVOS were developed. There are:

- **check_sms** — a script that tests Storage Management Service. First it generates local file of the size set in configuration and checks if the file with this name already exists on SMS, next uploads it to given SMS, downloads it and compares. With many conditions the script is able to inform administrator whether permission to folder is not right or whether file system is full. Additionally, the script shows average value of upload and download speed.

- **check_application** is the script to test applications installed in grid environment, especially scientific ones. It submits a job and waits for results. Then, it tests the results with a given logical conditions, where one can use functions such as `equals()`, `not_empty()` or `valid_pdf()` (i.e. `equals(#stdout,/etc/nagios/UNICORE/R/r_output.stdout) and valid_pdf(#plots.pdf)`). Output is compared to files generated in the installation process. This script works as a checker for applications (Blast, Clustal,

Fluent, R and CPMD are currently monitored) and also tests if the simple job can be run over UNICORE.

- **check_workflow** checks Workflow Service, Global SMS, Service Orchestrator an all UNICORE/X instances that Service Orchestrator can discover. It uploads a file to a Global SMS, submits a workflow, waits for the end of execution and checks if the output file has the same content as the the pattern file. The main idea is to create workflow description that has the same number of subtasks as available Target System Services (TSS) and executes command `cat` on every TSS to see if the whole UNICORE installation works properly. Additionally, the output of each workflow step is passed as an input for the next one. This test is the most important integration test. If it succeeds, no other test is executed. In most deployments this is the only test that is executed when UNICORE works properly.

- **check_gateway** uses UCC configuration file to connect to the UNICORE Gateway with HTTP client and gets all data in text format which can be analyzed. In configuration file user can specify what sites are required and if at least one of them is not registered in Gateway, the test will fail with appropriate message.

- **check_registry** executes `ucc system-info` command to gather the list of service containers registered in a UNICORE Registry. The script also provides option to specify Endpoint References that are required, and tests if every entry is available.

- **check_uvos** uses UVOS Command Line Client to execute `getMyIds` command and checks if there are any problems in connecting UVOS or if the result is good.

- **check_unicorex** uses Groovy script to check three conditions: if a Target System Factory (TSF) with a given name is available in a Global Registry, what services are in the TSF's Local Registry and how many TSSes are available (if none is found, TSS is created).

There are some tests that have not been developed yet. Such scripts are intended to be quite lightweight.

- **check_cis** connects to CIS and list CIPs that are managed,

- **check _cip** connects to CIS, get the data from requested CIP and compare them to the data gathered from CIP directly (via `ucc query-cip`),

- **check_servorch** submits work assignment to Service Orchestrator,

- **check_workflow_service** uses `ucc workflow-info` to check if Workflow Service is available and responds for a query.

Mentioned checkers are planned to be developed in the nearest future.

Apart from services, the monitoring system can gather additional data from UNICORE:

- additional plugin can *check free space on SMS* which is sent by Storage Management Service WS-Resource properties since UNICORE 6.3,

- in the recent UNICORE servers release every service advertises its version. Probe can check if every service is in a required version and *inform the administrator when software is out of date*,
- similarly the *versions of applications* installed on target systems may be checked.

The next place where monitoring can be used is internal state of the middleware server side. Every UNICORE module logs data using log4j library[10]. By checking the contents of the log files administrators can detect hard to otherwise identify problems. As effort to manually check all log files is very large, we provided a special log monitoring facility which can detect warnings and errors. The errors that are known to may happen can be filtered out. The utility is named LogWatcher and must be deployed on a server's machine. LogWatcher pushes problems to the Nagios server using using NSCA subsystem which allows for addition of so called passive tests to the framework. Passive tests are not invoked by Nagios but the status is updated externally, in our case by the LogWatcher.

The idea is to send information to Nagios with request to investigate the situation by administrator and to temporarily suspend checking of the log file with problem. After solving the problem administrator can resume LogWatcher.

The current version of LogWatcher also has additional abilities:

- it automatically adapts to changed `log4j.properties` file, so it can recognize different log4j entry formats,
- supports log4j file rolling,
- merges log entries that seem to be linked into a single error message.

To keep the system easy to use, an installation program that needs as little manual setting as possible was developed. The installer uses input provided by administrator to generate configuration files. However, whenever possible, it discovers information about the grid layout to minimize administrator effort. For instance if a Workflow Service is discovered in the configured Registry a complex workflow description testing all TSFs (as described in section 5) is automatically created.

6 Summary

The PL-Grid UNICORE monitoring system is able to supervise all UNICORE atomic services, the Workflow system and the UVOS server. Tests of other components are being developed and will be available in the same framework as the already existing probes. All tests are placed in the dependency graph to minimize monitoring overhead. When an error is detected more detailed tests are run and therefore the system is able to provide a detailed and reliable information on the failed component(s). The additional effort has been made to monitor grid applications functionality.

Except from developing aforementioned missing tests we will also work on improving tests execution intervals between dependent tests (there are some minor problems in Nagios with this) and on making the installation mechanism even easier and more flexible.

The system is deployed in PL-Grid project and proved to be extremely valuable for administrators. The constant monitoring allowed to identify few bugs in the UNICORE

middleware. The most significant one was the 3 months lifetime of the Workflow Factory Service, what obviously could not be detected by standard tests performed during development.

The whole monitoring infrastructure is available as open source under the BSD license at `http://unicore-life.svn.sourceforge.net/viewvc/unicore-life/monitoring/`

References

1. A. Streit: UNICORE: Getting to the heart of Grid technologies. eStrategies | Projects, 9th edition, British Publishers Ltd, Mar. 2009.
2. The UNICORE project website. `http://www.unicore.eu` (accessed on 13.06.2010)
3. The PL-Grid project `http://www.plgrid.pl` (accessed on 25.06.2010)
4. Ganglia Monitoring System `http://ganglia.sourceforge.net/` (accessed on 25.06.2010)
5. Zenoss website `http://www.zenoss.com/` (accessed on 25.06.2010)
6. OpenNMS website `http://www.opennms.org/wiki/Main_Page` (accessed on 25.06.2010)
7. M. Rambadt, M. Romberg *Simon 6 - Monitoring UNICORE 6 Resources* (CGW'09 Proceedings, Editors: Marian Bubak, Michal Turala, Kazimierz Wiatr; ACC CYFRONET AGH, Krakow, ISBN 978-83-61433-01-9, February 2010, pp. 134 - 142)
8. Nagios website `http://www.nagios.org/` (accessed on 25.06.2010)
9. Inca: User Level Grid Monitoring `http://inca.sdsc.edu/drupal/` (accessed on 25.06.2010)
10. Log4j library `http://logging.apache.org/log4j` (accessed on 25.06.2010)

Schriften des Forschungszentrums Jülich
IAS Series

1. **Three-dimensional modelling of soil-plant interactions: Consistent coupling of soil and plant root systems**
   by T. Schröder (2009), viii, 72 pages
   ISBN: 978-3-89336-576-0
   URN: urn:nbn:de:0001-00505

2. **Large-Scale Simulations of Error-Prone Quantum Computation Devices**
   by D. B. Trieu (2009), vi, 173 pages
   ISBN: 978-3-89336-601-9
   URN: urn:nbn:de:0001-00552

3. **NIC Symposium 2010**
   Proceedings, 24 – 25 February 2010 | Jülich, Germany
   edited by G. Münster, D. Wolf, M. Kremer (2010), v, 395 pages
   ISBN: 978-3-89336-606-4
   URN: urn:nbn:de:0001-2010020108

4. **Timestamp Synchronization of Concurrent Events**
   by D. Becker (2010), xviii, 116 pages
   ISBN: 978-3-89336-625-5
   URN: urn:nbn:de:0001-2010051916

5. **UNICORE Summit 2010**
   Proceedings, 18 – 19 May 2010 I Jülich, Germany
   edited by A. Streit, M. Romberg, D. Mallmann (2010), iv, 123 pages
   ISBN: 978-3-89336-661-3
   URN: urn:nbn:de:0001-2010082304

The UNICORE Grid technology provides a seamless, secure, and intuitive access to distributed Grid resources. UNICORE is a full-grown and well-tested Grid middleware system, which today is used in daily production worldwide. Beyond this production usage, the UNICORE technology serves as a solid basis in many European and International projects. In order to foster these ongoing developments, UNICORE is available as open source under BSD licence at www.unicore.eu.

The UNICORE Summit is a unique opportunity for Grid users, developers, administrators, researchers, and service providers to meet and share experiences, present past and future developments, and get new ideas for prosperous future work and collaborations. The UNICORE Summit 2010, the sixth in its series, took place 18 – 19 May at the Jülich Supercomputing Centre in Jülich, Germany.

The proceedings at hand include a selection of 14 papers that show the spectrum of where and how UNICORE is used and further extended.

This publication was edited at the Jülich Supercomputing Centre (JSC) which is an integral part of the Institute for Advanced Simulation (IAS). The IAS combines the Jülich simulation sciences and the supercomputer facility in one organizational unit. It includes those parts of the scientific institutes at Forschungszentrum Jülich which use simulation on supercomputers as their main research methodology.

IAS Series
Volume 5
ISBN 978-3-89336-661-3

JÜLICH
FORSCHUNGSZENTRUM