A Versatile Execution Management System for Next-Generation UNICORE Grids

Bernd Schuller, Roger Menday, and Achim Streit

Research Center Jülich, Central Institute for Applied Mathematics, Jülich, Germany
{b.schuller,r.menday,a.streit}@fz-juelich.de

Abstract. This paper builds on extensive experience with the UNICORE middleware to derive requirements for the next generation of Grid execution management systems. We present some well-known architectural ideas and design principles that allow building Grid servers that are adaptable to any type of target systems, from single workstations or PCs to huge supercomputers, and flexible enough for the novel usage scenarios and business models that are coming up in next-generation Grid systems. These ideas are used to implement an execution management system similar in scope to the UNICORE NJS.

1 Introduction

Compute resources available in present-day Grids range from small systems, such as single PCs, to very large systems such as supercomputers (for example in the DEISA project [1]) or PC farms as in EGEE[2].

These resources are made accessible through Grid middleware, specifically *execution management systems* (EMS). They serve a variety of functions in the areas authentication, authorisation and accounting (AAA), data management and execution management.

Grid execution management systems have to serve a wide range of compute resource capabilities, number of concurrent client connections, number of concurrent jobs, amount of data transferred and so on.

Additionally, in next-generation Grids new requirements are emerging [4]. In traditional scenarios such as scientific computing Grids, business rules such as billing or auditing procedures are simple, and usually hardcoded. However, business concerns such as service level agreements play an increasingly prominent role, as investigated for example in the NextGrid project[3]. To accommodate these needs, EMSs in next-generation Grids have to be highly flexible and reconfigurable.

The remainder of this paper is organised as follows: in the next section we present and review some experiences with the UNICORE Grid middleware made in the course of project and production use. From the capabilities and more importantly the shortcomings of this mature system, we derive a set of requirements for next-generation Grid servers. The remaining sections are devoted to design and partial implementation of a system called XNJS, respecting these requirements.

W. Lehner et al. (Eds.): Euro-Par 2006 Workshops, LNCS 4375, pp. 195–204, 2007.

2 Experiences with UNICORE

UNICORE, developed in the course of several German and European projects since 1997, 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 a graphical client and various server and target system components. The communication is based on a proprietary protocol using serialised Java objects (*abstract job objects, AJOs*). An overview on the history and usage scenarios is given in [6]. UNICORE is being used in various projects and production environments such as DEISA [1]. In the EU FP6 project UniGrids [7], it has evolved into a web services based Grid environment compliant with the web service resource framework (WSRF)[8], which is the prime candidate for realising the Open Grid Services Architecture (OGSA)[9] vision.

The UNICORE software is available open-source under a liberal, BSD-type license from the SourceForge repository [5].

The server side components of the current version of the UNICORE middleware (UNICORE 4) are organised into three tiers, the Gateway, NJS and TSI, that usually run on separate machines (Fig. 1).

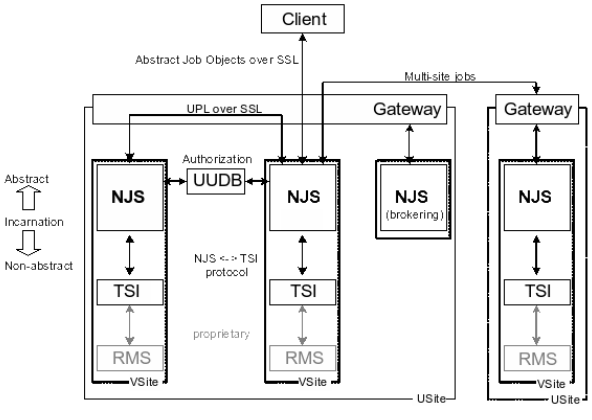

Fig. 1. The UNICORE 4 architecture

They serve distinct functions. The *gateway* is the primary point of entry, and can be considered a software firewall. It authenticates client requests and forwards them on to the next tier. The *target system interface* (TSI) is a stateless component talking directly to the underlying batch system. It offers a simple, text-based protocol to the batch system, and is used to execute scripts, submit batch jobs, request job status, get or write files and perform some common file system operations such as "list directory" or "copy". The main component in a UNICORE server installation is the *network job supervisor* (NJS), which will be discussed in detail in the next section.

2.1 The UNICORE NJS: A Gap Analysis

The central component of the UNICORE server side is the NJS (network job supervisor). The NJS is a multithreaded Java application that offers a variety of features, such as

- authorising users using the UNICORE user database (UUDB),
- translating the incoming abstract jobs into concrete jobs for the target system using a process called incarnation,
- submitting the concrete jobs to the TSI and monitoring their status,
- managing the outcome,
- communicating with the gateway,
- submitting sub-jobs to other Grid sites,
- keeping job state.

In UNICORE, abstract jobs can be arbitrarily complex and may involve workflows spanning multiple Grid sites. The NJS is a combination of a workflow processing engine and an execution management system for "atomic" jobs such as executing a script on the target system associated with the NJS.

Furthermore, the NJS offers several interfaces for add-on functionality such as brokering, resource reservation, and alternative file transfer mechanisms.

While the NJS (and thus UNICORE as a whole) offers a lot of functionality, there are some shortcomings as well, in the areas of flexibility, scalability and fault resilience.

Flexible processing and business rules. One limitation of the NJS is the fact that the processing rules as well as the business rules are hardcoded. Therefore, new requirements are only implementable by changing the core NJS code.

Currently, the processing rules are encoded into an object model, thus to add new types of actions or to modify the processing for certain types of action needs modifications of existing Java code.

Business rules are currently fixed as well (and mostly implicit). As an example scenario for the need for flexible business rules, one might think of different billing schemes based on the current user, such as pay-per-use for User A and a computing time budget for User B. Another business rule might be related to providing different resource views for different users. For example, User A should be able to use at most 10 nodes of a cluster, while User B should be allowed to use the full cluster. Currently, all users have the same set of available resources, and UNICORE relies on the underlying batch system to enforce policies such as the ones mentioned.

Scalability. As with any single software component, there are scalability issues with the NJS as well. There is no possiblity for clustering groups of NJSs. Furthermore, the current implementation of the NJS keeps a lot of state information in-memory, so during long-term operation, out of memory errors may occur.

Fault resilience. Fault resilience has many facets, but as an example scenario, consider the following. UNICORE was the Grid middleware of choice in the OpenMolGRID project [10][11], that sucessfully targeted Grid-based drug design. Often, complex multi-step jobs involving many Grid sites were run. However, sometimes job parts failed due to networking problems, or failure of one site, etc. This led to a failure of the whole job, and often to loss of results from other job parts, because the users did not take any precautions such as saving intermediary results. Here, the need for improved fault handling was felt, which should be based on a flexible set of business rules.

Limiting the scope. The UNICORE NJS does both atomic jobs (plain execution tasks) and multi-step, multisite workflows. We believe the workflow functionality should be provided elsewhere, in the interest of simplicity and modularity, and thus maintainability.

2.2 Requirements for a Next-Generation NJS

Analysing the experiences in using the current UNICORE implementation, we can identify a set of principles for designing a next-generation Grid execution management system. These can also be seen as non-functional requirements.

- *Highly modular, reconfigurable and scalable architecture:* The system must be composed of building blocks with well-defined functionality and well-defined responsibilities. These components must interact only using interfaces. The components must not make any assumptions about their environment.

  This ensures that an implementation of a subsystem can be replaced by another implementation without breaking other parts of the system.
- *No hardcoded processing rules:* The actions executed to run a job, or to transfer a file must be easy to modify, extend or even switch off. Additional processing steps (such as encryption / decryption) should be pluggable into the processing. It must be possible to add new types of actions without having to change the system core.
- *No hardcoded business rules:* There must be no static business rules in the system core. All business rules should be explicitly defined and configurable, ideally even on a per client request basis.
- *Scope limited to single-site actions:* The system should only deal with actions on a single site. Multi-site workflow functionality should be provided elsewhere.

A system that respects these principles will be easy to extend and adapt to new requirements.

To keep such a highly flexible system transparent and manageable, one has to take special care to give system administrators a detailed view on the current configuration, and to allow them to monitor the system closely.

3 The XNJS: Design and Implementation of a Next-Generation UNICORE NJS

In this section we outline the design and partial implemention of a Grid execution management system with we have named XNJS, that respects the basic requirements outlined in section 2.2.

A high-level depiction of a typical Grid node is shown in Fig. 2. The execution server resides in the " Grid tier".

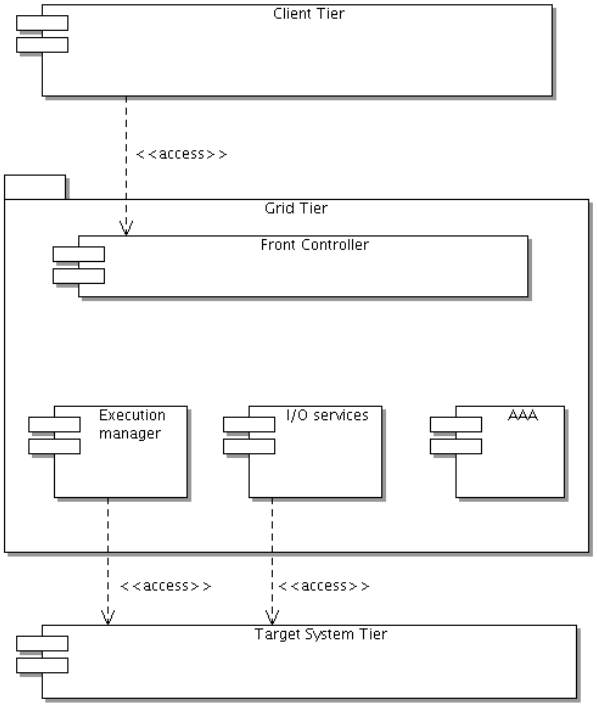

Fig. 2. Three-tier architecture representation of a Grid node

The front-controller subsystem takes care of the client communication. In present-day Grid systems this will usually be a set of WSRF compliant web services or components talking a proprietary protocol.

The services offered by the EMS can be grouped into execution services, file I/O services, and security services (authentication, authorization and accounting).

The actual resource, such as a batch system or database, is in the separate target system tier. Here, there are two scenarios: the target system and the EMS are running on the same machine, or, the EMS is running on a separate machine.

Additionally, there are aspects such as persistence, management, logging, etc., that are not shown in the figure but have to be taken into account in the design and implementation as well.

3.1 Core Architecture

It is well known that one fundamental principle for designing modular systems is the separation of interface from implementation [12]. When some component uses some other component of the system, it must not refer to a concrete implementation, but to the abstract interface of that service only.

For the XNJS, we have chosen an architecture similar to the "Microkernel" pattern [13] to maximise reconfigurability.

To realise this architecture, a component repository or component *container* is necessary, allowing storage and retrieval of concrete realisations of needed interfaces.

Using Java, this can be implemented using simple, lightweight containers such as the PicoContainer [16] or more complex frameworks such as Spring [17]. These containers offer convenient methods for dealing with component lifecycle, i.e. starting and stopping components. For our implementation we have chosen PicoContainer because of its easy embeddability and low memory footprint. Most services will in turn be dependent on other services. To resolve these dependencies, it is convenient to let the container take care of this task, and let it *inject* the dependencies using setter methods or constructor parameters[15]. A very important non-functional aspect of using dependency-injection is the improved testability of the individual components. For tests, "mock" dependencies can be used, allowing unit testing.

The actual runtime system configuration is defined at deployment time using configuration files.

3.2 Execution Management

In this section we focus on the execution management engine. Its functionality is the ability to execute a set of actions, keep track of action state, and offer some interfaces to the outside world for adding new actions and querying their status. Keeping such an engine flexible involves an extensible set of basic executable "building blocks" and a reconfigurable and extensible processing scheme for these executables.

Actions. Actions are the basic execution units in the XNJS. Actions are usually created within the front-end controller of the XNJS, and submitted for processing to the core execution engine. Their state chart is shown in Fig. 3.

If needed, the actions will communicate with the target system, invoke I/O services, start sub-actions etc.

The Action includes an XML work description, for example, a JSDL [18] document. The type of XML that is used defines the "type" of Action. The XNJS can be extended easily by adding support for new types of actions, specifically by adding processing code as outlined in the next section.

Processing Chains. The design of the processing itself should be flexible and adaptable to various deployment and usage scenarios. For this purpose, we have chosen to adopt a design pattern similar to the "chain of responsibility" from

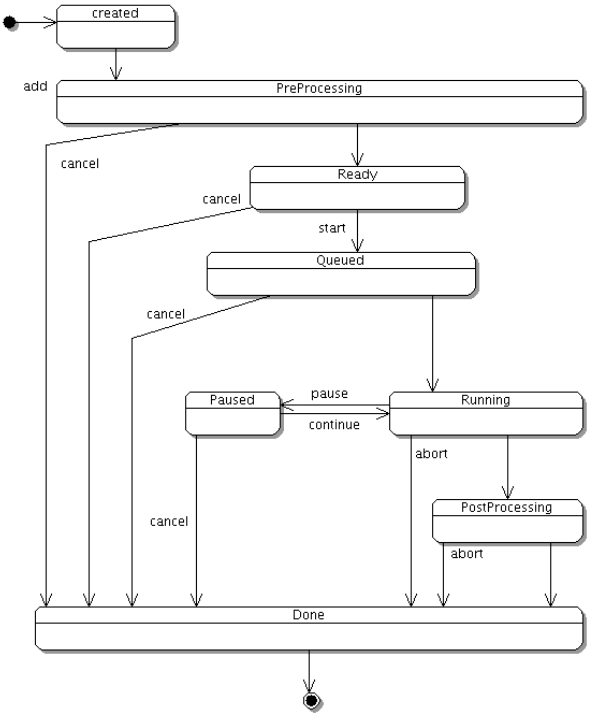

Fig. 3. The statechart for actions

[14]. The processing of an Action instance is done by a finite chain of processing elements (processors), that are called one after the other as depicted in Fig. 4. Each processor may perform arbitrary operations, change the action state, etc. Context information can be stored in the Action object itself that is passed along the processing chain. The processing chain for a given Action type is configurable, even at runtime if needed.

Processors can be used for any type of operation relevant during action execution, such as running an application, data encryption and decryption, , accounting and billing, or user notification. In this way, we realise the requirement that there are no hardcoded business or processing rules.

One disadvantage of this concept may be noted: processing can become quite complex, especially when different processors share context information, as is common during processing of workflows. Here, the processor responsible for workflow processing will start sub-actions, and will have to monitor these in order to decide when to start any dependent parts of the workflow.

3.3 Security

In Grids, many different trust and security policies are used, which may also change, thus it is vital that the EMS is flexible and does not have any hardcoded

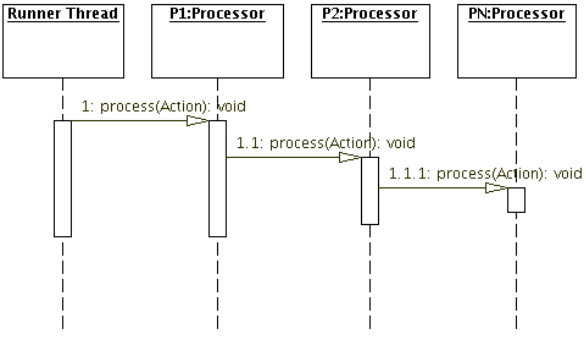

Fig. 4. A chain of processing elements

security settings. Thus, the core XNJS just provides a set of classes and patterns that can be used to build a solution that meets the security needs of the Grid it is deployed into.

Client information. A Client class can be used to capture information about the party that executes Actions on the XNJS, such as their name, authorization tokens. The Client object is usually generated in the front-end component, using information from the transport layer or the original message. This Client object is part of the Action during the Action's lifecycle.

Method call interception. We have employed the method call interception technique to allow very fine grained security checks. In the XNJS, the core method calls include authentication information in the form of a "Client" object. These method calls can be intercepted, and security checks can be executed. We have used aspect-oriented programming techniques [19] using AspectJ [20]. Rules and policies used to enforce security are pluggable, and easily extensible.

3.4 Status of the XNJS

The current status of the XNJS implementation is as follows.

Status of JSDL Processing. The most important functionality available in the XNJS is processing of atomic jobs, as defined by a JSDL document. The XNJS can execute JSDL jobs, including file staging in and out through local copy or HTTP. It supports important UNICORE concepts, such as abstract Application resources and abstract Filespaces (such as Root, Home, or Uspace).

Target System Support. Two target system interfaces currently exist. To support the use of the XNJS as "embedded" execution engine, a Java target system interface is available. This executes jobs locally by spawning a sub-process. Alternatively, an interface to a conventional Unicore 4.x TSI is available. Thus the XNJS can be used as execution management system for all those batch systems that can be accessed using Unicore 4.x, such as IBM LoadLeveler or PBS.

Management Interface. A running XNJS instance may be managed through the standard Java Management Extensions (JMX) interface. This allows to monitor the status of the Java virtual machine, to modify operational parameters, and to cleanly shutdown the XNJS.[21]

4 Conclusions and Outlook

Starting from an analysis of the Unicore NJS Grid execution server, we have derived some principles we believe to be indispensable for the next generation of Grid execution management servers. Using several well known design principles and patterns, we have designed and partly implemented a versatile, highly modular system that can be configured to suit various deployment needs and usage scenarios.

The use of a microkernel architecture with dependency injection allows easy configuration and simple testing and deployment of the system. The use of the "chain of responsibility" pattern within the execution engine allows building arbitrarily complex processing and business logic without modifying the core software.

The inherent flexibility and reconfigurability of the XNJS makes it useful in a variety of scenarios, for example

- as the backend behind a set of WSRF services implementing the UniGrids Atomic Services or OGSA-BES interfaces, with the XNJS embedded into the web services hosting,
- as a execution engine behind a web-application front end or a Representational State Transfer (REST)[22] interface,
- as part of a dynamic cluster of simple standalone worker nodes.

Future work will focus on ways to make the business rules of the system (including terms of use, billing, access rights and permissions) more explicit and dynamic.

References

1. DEISA: Distributed European Infrastructure for Supercomputing Applications http://www.deisa.org
2. EGEE: Enabling Grids for e-Science http://public.eu-egee.org/
3. NextGrid: Next-Generation Grids http://www.nextgrid.org
4. Third report of the "Next Generation Grids" Expert Group ftp://ftp.cordis.lu/pub/ist/docs/grids/ngg3_eg_final.pdf
5. UNICORE at SourceForge: http://unicore.sourceforge.net
6. A. Streit, D. Erwin, Th. Lippert, D. Mallmann, R. Menday, M. Rambadt, M. Riedel, M. Romberg, B. Schuller, and Ph. Wieder: UNICORE - From Project Results to Production Grids. L. Grandinetti (Edt.) "Grid Computing: The New Frontiers of High Performance Processing", pp. 357-376, Elsevier 2005

7. UniGrids homepage:
http://www.unigrids.org
8. Web Service Resource Framework:
http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsrf
9. The Open Grid Services Architecture, version 1:
http://www.ggf.org/documents/GFD.30.pdf
10. OpenMolGRID homepage:
http://www.openmolgrid.org
11. Dubitzky, W., McCourt, D., Galushka, M., Romberg, M., Schuller, B. Grid-enabled data warehousing for molecular engineering; Parallel Computing **30** (2004), 1019–1035
12. David L. Parnas: "On the criteria to be used in decomposing systems into modules", Communications of the ACM 15(2), Dec. 1972,1053-1058.
13. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P.: "A System of Patterns: Pattern-Oriented Software Architecture, Volume 1", Wiley, 1996
14. E. Gamma, R. Helm, R. Johnson, J. Vlissides: "Design Patterns", Addison-Wesley Publishing Company, 1995
15. Dependency Injection: http://www.martinfowler.com/articles/injection.html
16. PicoContainer:
http://picocontainer.codehaus.org
17. The Spring framework:
http://www.springframework.org
18. Job submission description language (JSDL):
http://forge.gridforum.org/projects/jsdl-wg/
19. Elrad, T., Filman, R.E., Bader, A.: "Aspect-oriented programming: Introduction", Communications of the ACM, **44** (2001), p. 29-32
20. AspectJ:
http://www.eclipse.org/aspectj
21. Java Management Extensions (JMX):
http://java.sun.com/products/JavaManagement
22. Fielding, R. Th.: "Architectural Styles and the Design of Network-based Software Architectures." Doctoral dissertation, University of California, Irvine, 2000.