

Redesigning the SEGL Problem Solving Environment: A Case Study of Using Mediator Components

Thilo Kielmann¹, Gosia Wrzesinska¹, Natalia Currle-Linde², and Michael Resch²

¹ Dept. of Computer Science, Vrije Universiteit, Amsterdam, The Netherlands
{kielmann|gosia}@cs.vu.nl

² High Performance Computing Center (HLRS), University of Stuttgart, Germany
{linde|resch}@hlrs.de

Abstract. The Science Experimental Grid Laboratory (SEGL) problem solving environment allows users to describe and execute complex parameter study workflows in Grid environments. Its current, monolithic implementation provides much high-level functionality for executing complex parameter-study workflows. Alternatively, using a toolkit of mediator components that integrate system-component capabilities into application code would allow to build a system like SEGL from existing, more generally applicable components, simplifying its implementation and maintenance. In this paper, we present the given design of the SEGL PSE, analyze the provided functionality, and identify a set of mediator components that can generalize the functionality required by this challenging application category.

1 Introduction

The SEGL problem solving environment [5] allows end-user programming of complex, computation-intensive simulation and modeling *experiments* for science and engineering. Experiments are complex workflows, consisting of domain-specific or general purpose simulation codes, referred to as *tasks*. For each experiment, the tasks are invoked with input parameters, that are varied over given parameter spaces, together describing individual parameter studies. SEGL allows users to program so-called *applications* using a graphical user interface. An application consists of several tasks, the *control flow* of their invocation, and the *data flow* of input parameters and results. For the parameters, the user can describe iterations for parameter sweeps; also, conditional dependencies on result values can be part of the control flow. Using such a user application program, SEGL can execute the tasks, provide them with their respective input parameters, and collect the individual results in an experiment-specific database.

In this paper, we revisit our view of component-based Grid application environments, present the given architecture of the SEGL PSE and its functionality, and identify a set of mediator components that can generalize the functionality

required by this challenging application category. An important insight is the requirement of a persistent application-execution service.

2 Component-based Grid application environments

A technological vision is to build Grid software such that applications and middleware will be united to a single system of components [3]. This can be accomplished by designing a toolkit of components that mediate between both applications and system components. The goal is to integrate system-component capabilities into application code, achieving both steering of the application and performance adaptation by the application to achieve the most efficient execution on the available resources offered by the Grid.

By introducing such a set of components, resources and services in the Grid get integrated into one overall system with homogeneous component interfaces. The advantage of such a component system is that it abstracts from the many software architectures and technologies used underneath. The strength of such a component-based approach is that it provides a homogeneous set of well-defined (component-level) interfaces to and between all software systems in a Grid platform, ranging from portals and applications, via mediator components to the underlying system software. A possible set of envisioned mediator components can be seen in Figure 1. We elaborate on them in the following.

Runtime Environment The runtime environment implements a set of component interfaces to various kinds of Grid services and resources, like job schedulers, file systems, etc. Doing so, the runtime environment provides an abstraction layer between application components and system components, while explicitly maintaining the application's security context. The interfaces are supposed to be implemented by a delegation mechanism that forwards invocations to service providers. Examples of such runtime environments are the GAT [1], or the platform as envisioned by GGF's SAGA-RG research group.

Steering Interface A dedicated part of the runtime environment is the steering interface. This component-level interface is supposed to make applications accessible by system and user-interface components (like portals, PSE's, or an application manager) like any other component in the system. One obvious additional use of such an interface would be a debugging interface for Grid applications.

Application-level meta-data repository This repository is supposed to store meta data about a specific application, storing, e.g., timing or resource requirements from previous, related runs. The collected information will be used by other components to support resource management (location and selection) and to optimize further runs of the applications automatically.

Application-level information cache

This component is supposed to provide a unified interface to deliver all kinds of meta-data (e.g., from a GIS, a monitoring system, from application-level

meta data) to the application. Its purpose is twofold. First, it is supposed to provide a unifying component interface to all data (independent of its actual storage), including mechanisms for service and information discovery. Second, this application-level cache is supposed to deliver the information really fast, cutting down access times of current implementations like Globus GIS (up to multiple seconds) to the order of a method invocation

Application steering and tuning components Controlling and steering of applications by the user, e.g. via application managers, user portals, and PSE's, requires a component-level interface to give external components access to the application. Besides the steering interface, also dedicated steering components will be necessary, both for mediating between application and system components, but also for implementing pro-active steering systems, carrying their own threads of activity.

Application Manager Component Such a component establishes a pro-active user interface, in charge of tracking an application from submission to successful completion. Such an application manager will be in charge of guaranteeing such successful completion in spite of temporary error conditions or performance limitations.

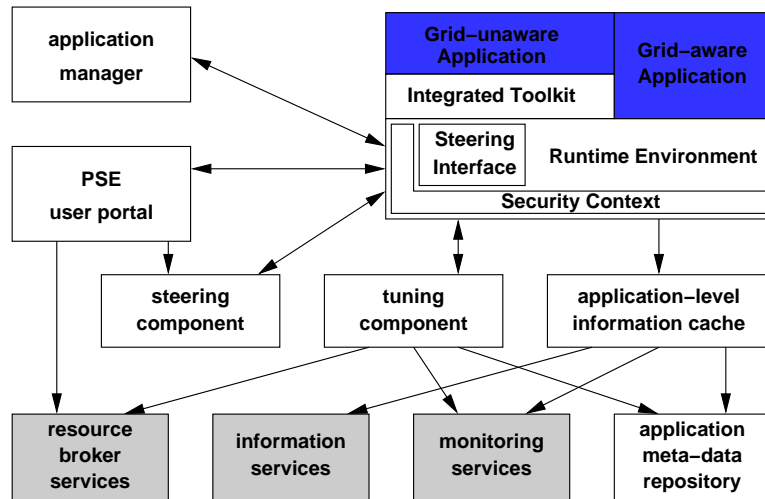


Fig. 1. Envisioned generic component platform

3 The SEGL system architecture

Figure 2 shows the current system architecture of SEGL. It consists of three main components: the User Workstation (Client), the Experiment Application

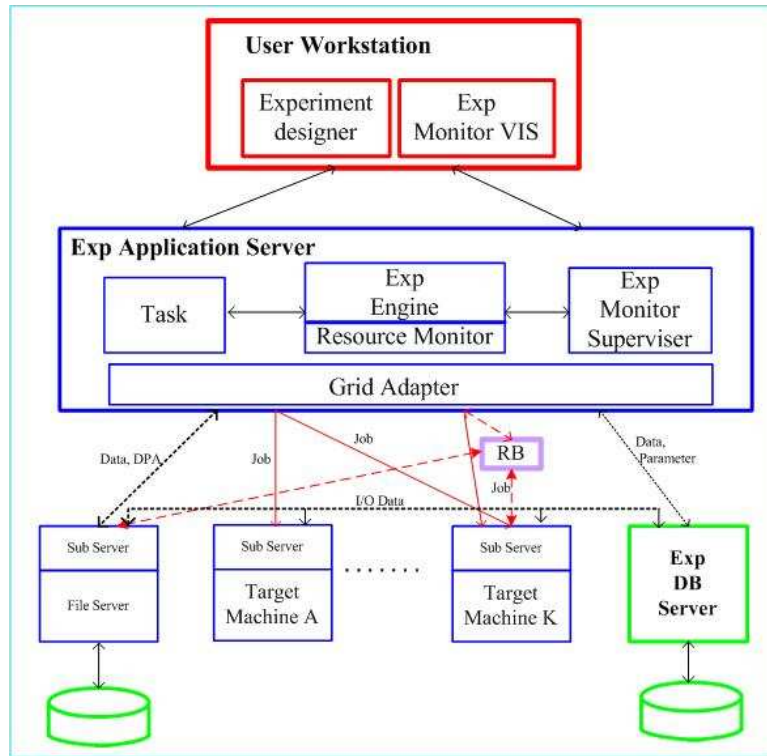


Fig. 2. Current SEGL architecture

Server (ExpApplicationServer), and the Experiment database server (ExpDB-Server). Client and ExpApplicationServer communicate with each other using a traditional client/server architecture, based on J2EE middleware. The interaction between ExpApplicationServer and the Grid resources is done through a Grid Adaptor, interfacing to Globus [6] and UNICORE [8] middleware.

The client on the user's workstation is composed of the graphical experiment designer tool (ExpDesigner) and the experiment process monitoring and visualization tool (ExpMonitorVIS). The ExpDesigner is used to design, verify and generate the experiment's program, organize the data repository and prepare the initial data, using a simple graphical language.

Each experiment is described at three levels: control flow, data flow and the data repository. The control flow level describes which code blocks will be executed in which order, possibly augmented by parameter iterations and conditional branches. Each block can be represented as a simple parameter study. An example is shown in Fig. 3. The data flow level describes the flow of parameter data between the individual code blocks. On the data repository level, a common

description of the metadata repository is created for the given experiment. The repository is an aggregation of data from the blocks at the data flow level.

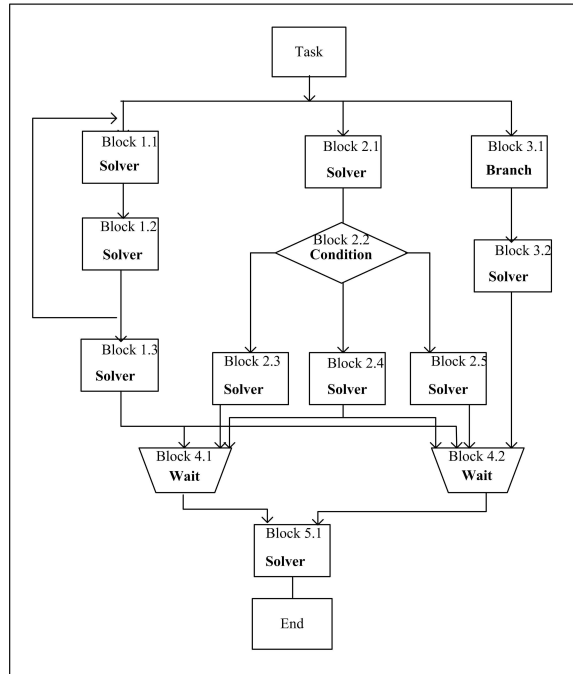


Fig. 3. Example experiment control flow

After completing the graphical design of the experiment program, it is “compiled” to the *container application*. This creates the experiment-specific parts for the ExpApplicationServer as well as the experiment’s data base schema. The container application of the experiment is transferred to the ExpApplicationServer and the schema descriptions are transferred to the server data base. Here, the meta data repository is created.

The ExpApplicationServer consists of the experiment engine (ExpEngine), the container application (Task), the controller component (ExpMonitorSupervisor) and the ResourceMonitor. The ResourceMonitor holds information about the available resources in the Grid environment. The MonitorSupervisor controls the work of the runtime system and informs the Client about the current status of the jobs and the individual processes. The ExpEngine is executing the application Task, so it is responsible for actual data transfers and program executions on and between server machine in the Grid.

The final component of SEGL is the data base server (ExpDBServer). The automatic creation of the experiment is done according to the structure designed

by the user. All data produced during the experiment such as input data for the parameter study, parameterization rules etc are kept in the ExpDBServer.

As SEGL parameter studies may run for significant amounts of time, application progress monitoring becomes necessary. The MonitorSupervisor, being part of the experiment application server, monitors the work of the runtime system and notifies the client about the current status of the jobs and the individual processes. The ExpEngine is the actual controller of the SEGL runtime system. It consists of three sub systems: the TaskManager, the JobManager and the DataManager. The TaskManager is the central dispatcher of the ExpEngine. It coordinates the work of the DataManager and the JobManager as follows:

1. It organizes and controls the execution sequence of the program blocks. It starts the execution of the program blocks according to the task flow and the conditions within the experiment program.
2. It activates a particular block according to the task flow, selects the necessary computer resources for the execution of the program and deactivates the block when this section of the program has been executed.
3. It informs the MonitorSupervisor about the current status of the program.

The DataManager organizes data exchange between the ApplicationServer and the FileServer and between the FileServer and the ExpDBServer. Furthermore, it provides the tasks processes with their the input parameter data. For progress monitoring, the MonitorSupervisor is tracking the status of the ExpEngine and its sub components. It forwards status update events to the ExpMonitorVIS, closing the loop to the user. SEGL's progress monitoring is currently split in to parts:

1. The experiment monitoring and visualization on the client side (ExpMonitorVIS). It is designed for visualizing the execution of the experiment and its computation processes. The ExpMontitorVis allows the user to start, stop, the experiment, and to change the input data and to subsequently re-start the experiment or some part of it.
2. The MonitorSupervisor within the application server controls and observes the work of the runtime system (Exp Engine). It sends continuous messages to the ExpMonitorVis on the client workstation.

This subdivision allows the user to disconnect from its running experiment. In this case, all status update messages will be stored with the application server for delivery to the client as soon as it will become reconnected.

4 Extracting mediator components from the SEGL functionality

The SEGL system constitutes an interesting use case for component-based Grid systems as it comprises all functionality required for complicated task-flow applications. In this section, we try to identify, within the existing SEGL implementation, generic functionality that could be implemented in individual, re-usable or exchangeable components.

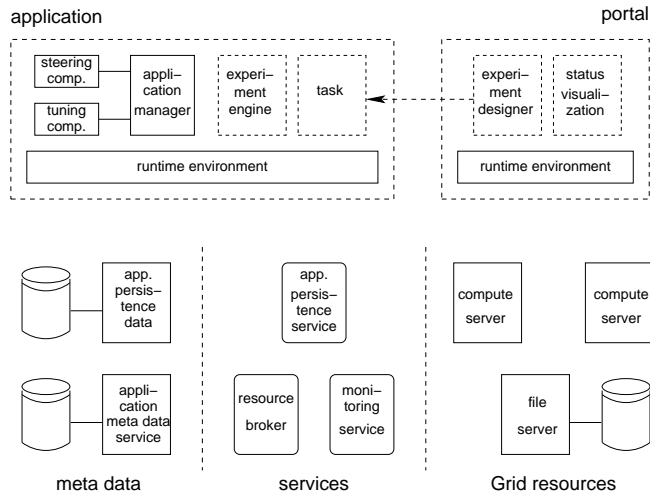


Fig. 4. SEGL redesigned using mediator components

In the current SEGL architecture, as shown in Fig. 2, there is a subdivision to three major areas: the user interface, the experiment application server, and the Grid resources and services. The latter consist of file servers for the experiment’s data, compute servers for experiment tasks, and additionally the experiment database, storing all experiment-specific status information. The user interface consists of the experiment programming environment (the “designer”) and the application execution visualization component.

By far the most interesting element of SEGL is the experiment application server. It concentrates the application logic (implemented via the *experiment engine* and the experiment-specific *task*), a Grid middleware interface layer (called *adaptor*), as well as progress monitoring functionality. Less visible in Fig. 2 is the fact that the experiment application server is a persistently running service. It has been designed as such to decouple the user interface from possibly long-running experiment codes.

Having such a persistently running service is certainly necessary to guarantee application completion in spite of transient error conditions, without user involvement. However, adding such a domain-specific, permanent service to the pre-installed middleware is causing administrative and security-related concerns.

Based on this analysis, we propose the following re-design based on mediator components, trying to refactor SEGL’s functionality into domain-specific components, complemented by general-purpose, reusable components. This redesign is shown in Fig. 4.

In this design, the software Grid infrastructure is organized in three tiers: resources, services, and meta data. For SEGL, relevant Grid resources are both compute and file servers, machines able to execute experimentation tasks and

providing the application data. These servers are accessible via Grid middleware, whichever happens to be installed on each resource.

Relevant Grid services are a resource monitoring service, like e.g. Delphoi [7] and a resource broker that matches tasks to compute servers. For the Grid services, we also propose an *application persistence service*. This is a persistent service, that keeps track of a given application and ensures it runs until successful completion, possibly restarting it in case of failures. Being a general-purpose, domain-independent service, it can be deployed in a virtual organization without overly administrative efforts, relying on a security concept that needs to be deployed only once for all kinds of applications. In a component-based architecture, we assume these services to have interfaces that fit into the component model.

The final infrastructure category is meta data. For persistent storage of such meta data, one or more servers can be deployed. One such component is the application meta data repository, equivalent to SEGL's current experiment data base. In addition, a meta data storage component is needed for the status information of the application persistence service.

The Grid infrastructure is used by two programs, the SEGL *application* and a user *portal*. Within these programs, Fig. 4 shows general-purpose components as solid boxes and domain-specific components as dashed boxes. Both programs are using the runtime environment for communication with the Grid infrastructure. The portal is implementing both the experiment designer as well as the experiment status visualization.

SEGL's monitoring and steering facilities also gets split across *application* and *portal*. Within the portal, the status visualization provides the user interface. Within the application, the *steering component* handles change requests for the parameter data. To allow the user to disconnect and later re-connect to his or her application, also the progress monitoring needs storage for its events that is persistent, at least until completion of the overall experiment. For this purpose, the *application meta data service* provides the appropriate storage facilities. The actual progress monitoring then takes place within the *application manager* component, but possibly a dedicated application monitoring and event handling component could be added.

The SEGL application is composed of components only. The experiment engine implements the SEGL-specific application logic, while the task component is created by the experiment designer within the SEGL portal. The experiment engine is accompanied by the generic application manager component which is responsible for both runtime optimization, using dedicated tuning and steering components, and for registering the SEGL application with the application persistence service. In the proposed combination, the experiment engine is responsible for the SEGL-specific control flow, while the application manager is in charge of all Grid-related control aspects, leading to a clear separation of concerns.

5 Conclusions

The SEGL problem solving environment allows end-user programming of complex, computation-intensive simulation and modeling *experiments* for science and engineering. As such, it constitutes an interesting use case for component-based Grid systems as it comprises all functionality required for complicated task-flow applications. In this paper, we have identified, within the existing, monolithic SEGL implementation, generic functionality that can be implemented in individual, reusable components. We have proposed a three-tier Grid middleware architecture, consisting of the resources themselves, persistent services, and meta data. The necessity of a persistent application-execution service was an important insight. Based on this architecture, we were able to compose a SEGL experiment execution application from mostly general-purpose components, augmented only by a SEGL-specific experiment engine and the dynamically created experiment task description. With this architecture we tried to refactor a system like SEGL such that general-purpose functionality is implemented in reusable components while a minimal set of domain-specific components can be added to compose the overall application.

With currently available technology, such components do not exist yet, as suitable component models, and especially generally accepted and standardized interfaces, are subject to ongoing work [4]. Once such components become available [2], refactoring SEGL's implementation will be an interesting exercise.

References

1. G. Allen, K. Davis, T. Goodale, A. Hutanu, H. Kaiser, T. Kielmann, A. Merzky, R. van Nieuwpoort, A. Reinefeld, F. Schintke, T. Schütt, E. Seidel, and B. Ullmer. The Grid Application Toolkit: Towards Generic and Easy Application Programming Interfaces for the Grid. *Proceedings of the IEEE*, 93(3):534–550, 2005.
2. CoreGRID Virtual Institute on Problem Solving Environments, Tools, and GRID Systems. Proposal for mediator component toolkit. CoreGRID deliverable D.ETS.02, 2005.
3. CoreGRID Virtual Institute on Problem Solving Environments, Tools, and GRID Systems. Roadmap version 1 on Problem Solving Environments, Tools, and GRID Systems. CoreGRID deliverable D.ETS.01, 2005.
4. CoreGRID Virtual Institute on Programming Models. Proposal for a Common Component Model for GRID. CoreGRID deliverable D.PM.02, 2005.
5. N. Currle-Linde, U. Küster, M. Resch, and B. Risio. Science Experimental Grid Laboratory (SEGL) Dynamical Parameter Study in Distributed Systems. In *ParCo 2005*, Malaga, Spain, 2005.
6. I. Foster and C. Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *Int. Journal of Supercomputer Applications*, 11(2):115–128, 1997.
7. J. Maassen, R. V. van Nieuwpoort, T. Kielmann, K. Verstoep, and M. den Burger. Middleware Adaptation with the Delphoi Service. *Concurrency and Computation: Practice and Experience*, 2006. Special issue on Adaptive Grid Middleware.
8. D. Erwin (Ed.). *Joint Project Report for the BMBF Project UNICORE Plus*. UNICORE Forum e.V., 2003.