A Light-Weight Grid Workflow Execution Engine Enabling Client and Middleware Independence*

Erik Elmroth, Francisco Hernández, and Johan Tordsson

Dept. of Computing Science and HPC2N
Umeå University, SE-901 87 Umeå, Sweden
{elmroth, hernandf, tordsson}@cs.umu.se

Abstract. We present a generic and light-weight Grid workflow execution engine made available as a Grid service. A long-term goal is to facilitate the rapid development of application-oriented end-user workflow tools, while providing a high degree of Grid middleware-independence. The workflow engine is designed for workflow execution, independent of client tools for workflow definition. A flexible plugin-structure for middleware-integration provides a strict separation of the workflow execution and the processing of individual tasks, such as computational jobs or file transfers. The light-weight design is achieved by focusing on the generic workflow execution components and by leveraging state-of-the-art Grid technology, e.g., for state management. The current prototype is implemented using the Globus Toolkit 4 (GT4) Java WS Core and has support for executing workflows produced by Karajan. It also includes plugins for task execution with GT4 as well as a high-level Grid job management framework.

1 Introduction

Motivated by the tedious work required to develop end-user workflow tools and the lack of generic tools to facilitate such development, this contribution focus on a light-weight and Grid-interoperable workflow execution engine made available as a Grid service. As a point of departure, we identify important and generic capabilities supported by well-recognized complete workflow systems [18, 15, 9, 14, 1, 10, 2] (e.g., workflow design, workflow repositories, information management, workflow execution, workflow scheduling, fault tolerance, and data management). However, many of these projects provide similar functionality and much work is overlapping, as the systems have been developed independently [18].

The tool presented here is not proposed as an alternative to these more complete workflow systems, but as a core component for developing new end-user tools and problem solving environments. The aim is to offer a generic workflow

* This research was conducted using the resources of the High Performance Computing Center North (HPC2N). Financial support has been provided by The Swedish Research Council (VR) under contract 621-2005-3667.

execution engine that can be employed for building new high-level tools as well as to provide support for both processing individual tasks on multiple Grid middlewares and accepting different workflow languages as input. The engine is light-weight as it focuses only on workflow execution (i.e., selecting tasks that are ready to execute) and its corresponding state management.

The engine is developed with a strict focus on Grid resources for task processing and makes efficient use of state-of-the-art Web and Grid services technology. The current prototype is implemented using the Java WS Core from the Globus Toolkit 4 (GT4) [7]. The service has support for executing workflows expressed either in its native workflow language or the Karajan [16] format. It includes plugins for arbitrary Grid tasks, e.g., for execution of computational tasks in GT4 and in the high-level Grid Job Management Framework (GJMF) [3, 5], as well as GridFTP file transfers.

2 System and Design Requirements

The general system requirements follow directly from the aim and motivation for the proposed workflow engine. As it is developed with a general aim to provide an efficient and reusable tool for managing workflows in Grid environments, overall requirements include client and middleware independence, modularity, customizability, and separation of concerns [4]. A set of high-level design requirements for Grid workflow systems includes the following.

- The *workflow execution* should be separated from the *workflow definition*. The former must be done by the engine, the latter can be done, e.g., by an application specific GUI or a Web portal. Furthermore, workflow repositories and application specific information should not be managed by the service.
- The workflow engine should be independent of the Grid middleware used to execute the tasks, with middleware-specific interactions performed by plugins. The plugins should in turn be unaware of the context (the workflow) to which individual Grid jobs belong.
- The design can and should to a large extent leverage state-of-the-art Grid technology and emerging standards, e.g., by making use of general features of the Web Services Resource Framework (WSRF) [13] instead of implementing their workflow-specific counterparts.
- The engine should have a clean separation between the state management and the handling of task dependencies.

In addition to the high-level design requirements, the following specific system requirements are highlighted. The workflow system should:

- provide support for executing workflows, managing workflow state, and pausing and resuming execution. This enables restart of partially completed workflows stored on disk, and provides a foundation for fault tolerant workflow execution.

- provide support for both abstract (resources unspecified) and concrete workflows (resources specified on a per-task level) as well as arbitrary nestings of workflows.
- provide support for *dynamic workflows*, i.e., making it possible to modify an already executing workflow, by pausing the execution before modification.
- provide support for workflow monitoring, both synchronously and by asynchronous notifications.
- provide support for notifications of different granularity, e.g., enabling asynchronous status updates on both a per workflow and a per task basis.

These requirements are in agreement with and extend on the requirements of Grid workflow engines presented in [6]. How the requirements are mapped to the actual implementation is presented in Section 3.

3 Design and Implementation

The design requirements of customizability and ability for integration with different client tools and middlewares are met by use of appropriate plugin points. The chain-of-responsibility design pattern allows concurrent usage of multiple implementations of a particular plugin. The three main responsibilities of the workflow service, namely management of task dependencies (i.e., deciding the task execution order), execution of workflow tasks on Grid resources, and management of workflow state, are each performed by separate modules.

Reuse, in a broad sense, is a key issue in the design. The workflow service reuses ideas from an architecture for interoperable Grid components [5] and builds on a framework for managing stateful Web services and notifications [13]. Exploiting the capabilities offered by GT4 Java WS Core (e.g., security and persistency) also simplifies the design and implementation of the service.

3.1 Modelling Workflows with the WSRF

The workflow engine uses the tools provided by GT4 Java WS Core to make the engine available as a Grid service and to manage the workflow state. Building the engine on top of Java WS Core should not be interpreted as built primarily for GT4-based Grids. Integration with different middlewares is provided by middleware-specific plugins which are independent from the workflow execution.

By careful design, the service can handle arbitrarily many workflows concurrently without these interfering with each other. Multiple users can share the same workflow service, but only the creator of a workflow instance can monitor and control that workflow. Each workflow is modelled as a WS-Resource and all information about a workflow, including task descriptions, inter-task dependencies and workflow state, is stored as WS-ResourceProperties. The default behavior is to store each WS-Resource in a separate file, although alternative implementations such as persistency via database can be added easily. Reuse of

the Java WS Core persistency mechanisms makes workflow state handling trivial. Workflow state management enables the control of long-running workflows and the recovery of workflows, e.g., upon service failures.

The states handled include `default`, `ready`, `running`, and `completed`, which apply to both tasks and (sub)workflows. Tasks can also be `failed` whereas workflows can be `disabled`. All newly created tasks and workflows have the default state. A task/workflow is ready to be started when all tasks on which it depends are completed. Running tasks are processed by some Grid resource until they become either completed or failed. A running workflow has at least one task that is not completed and no failed task, whereas completed workflows only contain completed tasks/subworkflows. A workflow becomes disabled either if a task fails or if the user requests the workflow to be paused. No new tasks are initiated for disabled workflows. A resume request from the user is required to make a disabled workflow running again. If the workflow becomes disabled due to task failure, the user must modify the workflow (to correct the failed task) before issuing the resume request.

3.2 Architecture of the Workflow Engine

The workflow service implements operations to (i) create a new workflow, (ii) suspend the execution of a workflow, (iii) resume execution of a workflow, (iv) modify a workflow, and (v) cancel a workflow. The service also supports monitoring of workflows, either by explicit status requests or by asynchronous notifications of updates. To support a wide range of client requirements, different granularities of notifications are available, ranging from a single message upon workflow completion to detailed updates every time a task changes its state. As Java WS Core contains mechanisms for managing WS-Resources (in this case workflows), the monitoring functionality as well as operations (iv) and (v) are trivial to implement (using WS-Notifications [8], and WSRF [13], respectively).

The architecture of the workflow engine is shown in Figure 1. User credentials are delegated from clients to the workflow service to be used when interacting with Grid resources. This requires the *Web service interface* to perform authentication and authorization of clients. All incoming requests are forwarded to the *Coordinator*, which organizes and manages the execution of tasks (and subworkflows) in the workflow and handles workflow state. When a new workflow is requested, the Coordinator uses the *Input Converter* plugin(s) to translate the input workflow description from the native format specified by the client to the internal workflow language. However, the Input Converters do typically not translate the individual task descriptions, as these are only to be read by the *Grid Executor* plugin(s), which the Coordinator invokes to process one (or more) tasks.

The Grid Executor interface defines operations to initiate new tasks, to reconnect to already initiated tasks after service restart, and to cancel tasks, corresponding to the create, resume and cancel operations in the Web service interface. There is however no operation to pause a running task, as this functionality generally is not supported by Grid middlewares. Computational Grid Executors

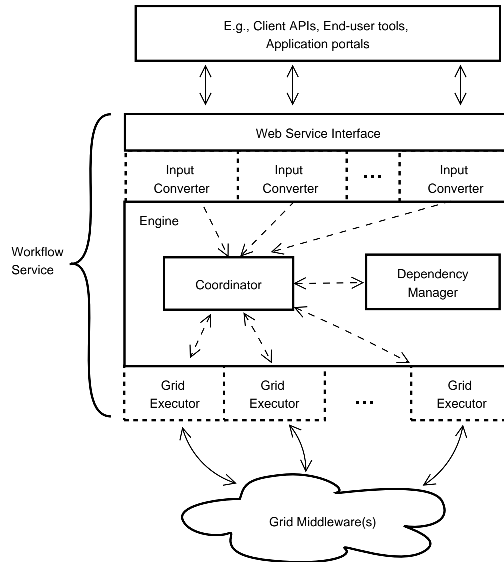

Fig. 1. Overview of the workflow service architecture.

also ensure that tasks' input and output files are transferred in compliance with the data dependencies in the workflow, but are unaware of the context (the workflow) to which each task belongs. This type of Grid Executor only requires a basic job submission mechanism, e.g., WS-GRAM [7], but can also make use of sophisticated frameworks, e.g., the GJMF [3] for resource brokering and fault tolerant job execution, should such functionality be available. Scheduling is performed on a per-task basis by the Grid Executors plugins. However, tools for planning or pre-scheduling of workflows (e.g., Pegasus [2]) can be employed if such functionalities are required. Moreover, support for abstract and concrete workflows is granted via the Executor plugins and external tools respectively.

Before the Coordinator can invoke the Grid Executor(s) in order to start new tasks, the *Dependency Manager* is used to select which task(s) to execute. This module keeps track of dependencies between tasks (and subworkflows) in a workflow, and determines when a task (or subworkflow) is ready to start. The Coordinator invokes the Dependency Manager to get a list of tasks available for execution when a new workflow is started, when a task in an existing workflow completes, and when a paused workflow is resumed.

3.3 Properties of the Workflow Language

In the workflow service, workflows are described in a data flow language, defined using XML schema. In this language, users specify task dependencies, not task execution order. This removes the burden of figuring out which tasks can execute in parallel as this is the responsibility of the Dependency Manager.

The workflow language supports arbitrary nesting of tasks and subworkflows within a workflow. Each task (or workflow) specifies a set of input and output

ports. A (sub)workflow contains a set of *links*, where each link connects an output port of one task/workflow with an input port of another. The task description contains a field to specify how to perform the task. By having this field generic, the usage of multiple Grid task description formats is possible. Different formats for individual task descriptions may even be used within the same workflow. This design also enables support for new task types, e.g., database queries and Web service invocations, to be added by implementing Grid Executor plugins rather than extending the workflow language.

4 Analysis and Comparison with other Systems

One of the main objectives of this work is to provide independence not only of Grid middleware but also of input representation, the latter achieved by converters that translate different workflow languages to the service's internal data flow language. How difficult these translations are depend on the style of the original client's language and the amount and type of information that can be expressed in that language. Data flow languages with similar input/output port structure are simple to translate. *Control flow* languages can also be translated by specifying ports that represent flow of control rather than data transfers. For example, the subset of the Karajan language [16] that performs basic interactions with Grid resources (job submissions, file transfers, and sequential and parallel definition of tasks) has been translated as described above.

Petri net languages pose more difficulties. Places and transitions representing data flows can easily be translated to the service's internal data flow language. However, there is not an equivalent concept for representing loops in the service's language. Finally, it can also be hard to translate languages that do not have all the information encoded in the workflow description but rely on the runtime system to obtain the missing information (e.g., a workflow system that dynamically queries a repository to obtain the input/output structure of workflow tasks).

While several workflow projects have been built to interact with Grid systems [15, 14, 1], many of them have not been designed for exclusive use of Grid resources for workflow execution. Nevertheless they are integrated solutions with sophisticated graphical environments, workflow repositories, and fault management mechanisms. Our work does not attempt to replace those systems, but to provide a means for accessing advanced capabilities offered by multiple Grid middlewares. These benefits are obtained by the separation of the workflow execution from its definition and by making use of well-established protocols. Furthermore, implementing the workflow engine as a stateful WSRF service facilitates the management and control (including fault recovery) of long-running workflows which are common in Grid computing.

The P-GRADE portal [12] and Karajan [16] also focus on the use of resources from different Grids within the same workflow. P-GRADE offers a collaborative environment in which multiple users define workflows through a client application, and control and manage workflows through a portal. The workflows can access resources from multiple Globus-based virtual organizations. Our work

goes beyond this functionality by adding the capability of using other middlewares besides Globus and also offering independence of input language. Karajan also provides a level of interoperability between different execution mechanisms (mainly GT2, GT4, Condor, and the SSH protocol) through the use of providers that allow selection of middleware at runtime. However, while Karajan has a stronger focus on the interaction between users and workflows, our work focuses on handling the workflow state, delegating the interaction with users to clients that have access to the workflow service.

There are a few projects that are using WSRF to leverage the construction of workflow services. The Grid Workflow Execution Service (GWES) [11] uses a Petri net language to define and control Grid workflows. Besides the differences in workflow language type, the main difference between GWES and our work is the ability of using multiple input representations offered by our contribution. The Workflow Enactment Engine Project (WEEP) [17] provides a BPEL engine for Grid Workflows. The engine is accessible as a WSRF service running in a GT4 container. However, WEEP is focused on Web service invocations and not on interfacing with Grid middleware.

5 Concluding Remarks

The goal of this research is to investigate how to design a light-weight workflow engine that can be reused by different high-level tools. General requirements for portability and interoperability are supported by the use of an appropriate plugin-structure for workflow language formats and for interacting with different Grid middlewares. Scalability is obtained by handling multiple workflows and by supporting large hierarchical workflows. The workflow service performs monitoring, state management, fault recovery, and it uses appropriate security mechanisms to achieve user isolation. The Executor plugins handle data movement, job submission, information retrieval, and just-in-time scheduling. External tools can be employed for planning and pre-scheduling of workflows. We finally note that much of the supported functionality is obtained with little or no effort by appropriate use of the WSRF.

6 Acknowledgements

We thank P-O Östberg for fruitful discussions on workflow system design and language constructs, and for collaboration in the integration of the GJMF [3]. We are also grateful to the anonymous referees for their constructive comments.

References

1. I. Altintas, A. Birnbaum, K. Baldridge, W. Sudholt, M. Miller, C. Amoreira, Y. Potier, and B. Ludaescher. A framework for the design and reuse of Grid workflows. In P. Herrero et al., editors, *Intl. Workshop on Scientific Applications on Grid Computing (SAG'04),* LNCS 3458, pages 119–132. Springer-Verlag, 2005.

2. E. Deelman, G. Singh, M. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G.B. Berriman, J. Good, A. Laity, J.C. Jacob, and D.S. Katz. Pegasus: a framework for mapping complex scientific workflows onto distributed systems. *Scientific Programming*, 13(3):219–237, 2005.
3. E. Elmroth, P. Gardfjäll, A. Norberg, J. Tordsson, and P-O. Östberg. Designing general, composable, and middleware-independent Grid infrastructure tools for multi-tiered job management. In T. Priol and M. Vaneschi, editors, *Towards Next Generation Grids*, pages 175–184. Springer-Verlag, 2007.
4. E. Elmroth, F. Hernández, J. Tordsson, and P-O. Östberg. Designing service-based resource management tools for a healthy Grid ecosystem. In R. Wyrzykowski et al., editors, *Parallel Processing and Applied Mathematics. 7th Int. Conference, PPAM 2007*. Lecture Notes in Computer Science, Springer-Verlag, 2007 (to appear).
5. E. Elmroth and J. Tordsson. An interoperable, standards-based Grid resource broker and job submission service. In H. Stockinger et al., editors, *First International Conference on e-Science and Grid Computing*, pages 212–220. IEEE CS Press, 2005.
6. S. Eswaran, D. Del Vecchio, G. Wasson, and M. Humphrey. Adapting and evaluating commercial workflow engines for e-Science. In *Second IEEE International Conference on e-Science and Grid Computing*. IEEE CS Press, 2006.
7. I. Foster. Globus toolkit version 4: Software for service-oriented systems. In H. Jin et al., editors, *IFIP International Conference on Network and Parallel Computing,* LNCS 3779, pages 2–13. Springer-Verlag, 2005.
8. S. Graham, D. Hull, and B. Murray. Web Services Base Notification 1.3 (WS-BaseNotification). http://docs.oasis-open.org/wsn/wsn-ws_base_notification-1.3-spec-os.pdf, May 2007.
9. Z. Guan, F. Hernández, P. Bangalore, J. Gray, A. Skjellum, V. Velusamy, and Y. Liu. Grid-Flow: a Grid-enabled scientific workflow system with a petri-net-based interface. *Concurrency Computat.: Pract. Exper.*, 18(10):1115–1140, 2006.
10. F. Hernández, P. Bangalore, J. Gray, Z. Guan, and K. Reilly. GAUGE: Grid Automation and Generative Environment. *Concurrency Computat.: Pract. Exper.*, 18(10):1293–1316, 2006.
11. A. Hoheisel. User tools and languages for graph-based Grid workflows. *Concurrency Computat.: Pract. Exper.*, 18(10):1101–1113, 2006.
12. P. Kacsuk and G. Sipos. Multi-grid and multi-user workflows in the P-GRADE Grid portal. *J. Grid Computing*, 3(3-4):221–238, 2006.
13. OASIS. OASIS Web Services Resource Framework (WSRF) TC. http://www.oasis-open.org/committees/wsrf/, May 2007.
14. T. Oinn, M. Addis, J. Ferris, D. Marvin, M. Senger, M. Greenwood, T. Carver, K. Glover, M.R. Pocock, A. Wipat, and P. Li. Taverna: A tool for the composition and enactment of bioinformatics workflows. *Bioinformatics*, 20(17):3045–3054, 2004.
15. I. Taylor, M. Shields, I. Wang, and A. Harrison. The Triana workflow environment: architecture and applications. In I. Taylor et al., editors, *Workflows for e-Science*, pages 320–339. Springer-Verlag, 2007.
16. G. von Laszewski and M. Hategan. Workflow concepts of the Java CoG Kit. *J. Grid Computing*, 3(3–4):239–258, 2005.
17. WEEP. The Workflow Enactment Engine Project. http://weep.gridminer.org, May 2007.
18. J. Yu and R. Buyya. A taxonomy of workflow management systems for Grid computing. *J. Grid Computing*, 3(3–4):171–200, 2006.