A PRACTICAL APPROACH FOR A WORKFLOW MANAGEMENT SYSTEM

Simone Pellegrini, Francesco Giacomini, Antonia Ghiselli
INFN Cnaf
Viale Berti Pichat, 6/2 - 40127 Bologna, Italy
simone.pellegrini@cnaf.infn.it
francesco.giacomini@cnaf.infn.it
antonia.ghiselli@cnaf.infn.it

Abstract A variety of grid middlewares and workflow languages causes the existence of many workflow management systems (WfMS). Formalisms used to represent workflows vary from simple Directed Acyclic Graphs (DAG) to more complex (non deterministic) Petri Nets. Therefore a workflow description is strictly bound to a particular WfMS and to the computational resources that WfMS address, as far as no cooperation among WfMSs exists. This might be critical in scientific workflows where a large amount of resources is usually needed. In this paper we propose a WfMS that aims at language independence and Grid middleware abstraction dealing with interoperability as proposed in the reference model of the Workflow Management Coalition (WfMC). The main goal of such WfMS is to provide an effective solution to run complex scientific workflows (legacy or not) taking full advantage of the distributed and etherogeneous nature of the Grid. A Petri Net formalism has been chosen as internal representation due to its formal behavioral description and the existence of several analysis tools. Our proposed WfMS will be implemented on top of the gLite Grid middleware provided by the EGEE project because of its stability and large adoption.

Keywords: workflow management, grid computing, interoperability, Petri Net, EGEE/gLite

1. Introduction

The evolution of the Grid towards a *service-oriented* architecture enables scientists to build complex applications as *workflows*. WfMSs allow the composition and execution of such distributed applications at a high abstraction level; a workflow language, usually graph based, is used to specify dependencies (control and data flow) between tasks. Several WfMSs exist both in scientific and in business environments. This underlines the research interest in this field.

Unlike business WfMSs, scientific ones lack a recognized standard; as a consequence several workflow languages exist. Apart from the syntax, these languages differ for the formalism used to express the workflow model. Most of the graphical workflow languages are based on DAGs where the *control flow* can be described in terms of *sequence*, *parallelism* and *choice*. More powerful than DAGs, formalisms such as *Petri Nets* and π-*Calculus* allow to define *iteration* (also know as *loop* or *cycle*). As a consequence of that variety of languages and formalisms, WfMSs are *incompatible*. Furthermore a WfMS usually address a small set of computational resources and without interoperability scientific workflows cannot fully take advantage from the distributed, heterogeneous nature of the Grid. The Workflow Management Coalition (WfMC) encourages WfMSs standardization in its *reference model* which defines a set of APIs (called WAPI) and interfaces numbered from 1 to 5 in order to achieve *interoperability*. In particular, interface 4 describes different levels of workflow coordination/cooperation. Unfortunately the WfMC has so far failed its standardization scope and no WfMS formally follows its reference model.

In this paper we propose a *generic* WfMS architecture that abstracts from the underlying grid middleware and deals with workflow interoperability. As we will see in detail, the definition of a grid abstraction layer makes it possible to build a middleware-independent workflow engine. A Petri Net formalism is used as the internal representation due to its *formal semantics*. Petri Nets capture both the control and data flow of the workflow, they formally describe its state evolution and they are Turing-complete. Workflow interoperability is addressed using language translators and model converters.

The implementation of our WfMS will rely on the gLite Grid middleware. gLite exposes several Grid *services* with a good level of reliability and the amount of managed resources allows users to execute complex and large workflows. The Job Description Language (JDL) is the *lingua-franca* of the gLite middleware, it is used for job and also workflow (expressed as DAGs) descriptions. DAGs are executed by Condor DAGMan [1]which provides a *basic* support for workflow management. In fact, DAGMan practically lacks failure recovery and that limits expressiveness in workflow design. With this work we want to improve workflow support in gLite allowing users to keep all the advantages of using a full feature WfMS.

In Sect. 2 we will go along describing in details at the engine architecture, defining the layers and how they interact. In Sect. 3 we will focus on the interoperability problem and in Sect. 4 we will show some details related to the implementation of such WfMS, concluding with the description of how we intend to progress with this work in the future in Sect. 5.

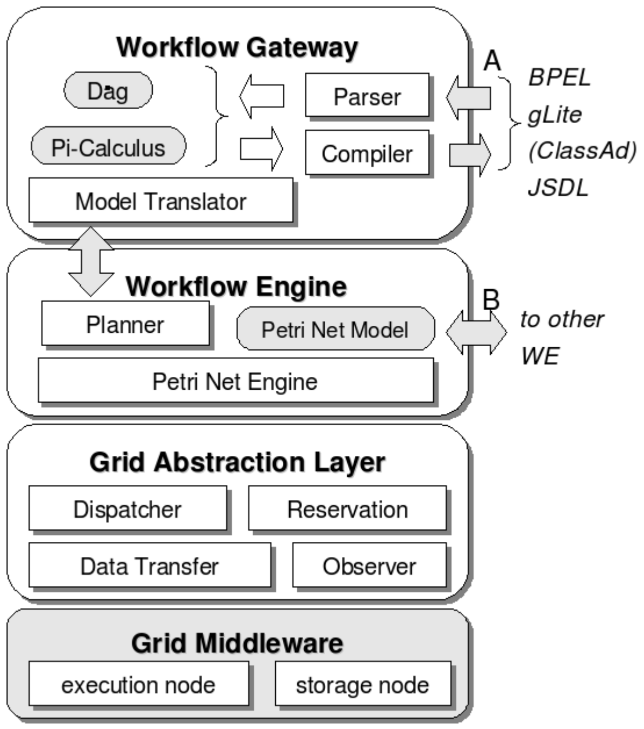

Figure 1. The WfMS Architecture, interfaces *A* and *B* will be explained in more detail in Sect. 4.

2. Workflow Architecture Overview

In this section we propose a generic WfMS architecture that aims at Grid middleware *independence*, as proposed in [2], and at *multi-language* support. The use of a layered architecture makes it possible to abstract both from a particular Grid infrastructure and a workflow language in order to provide portability and multi-language compatibility.

An outline of such architecture is shown in Fig. 1. At the bottom lies the basic Grid infrastructure: a collection of *computational* and *storage* resources. These resources are transparent to users thanks to a so called *Grid middleware* which acts as a mediator that provides a consistent and homogeneous access to them.

Since *multiple* Grid infrastructures still exist, a *Grid Abstraction Layer* is introduced in order to abstract high level Grid functionalities such as job submission, data transfer, job state observation and resource reservation. This makes it possible to *decouple* the workflow engine from the underlying grid architecture allowing workflows to use a large set of Grid infrastructures and therefore resources.

The workflow engine is the main component of the WfMS; it basically submits *tasks* to the Grid taking care of their dependencies and the overall workflow execution. The engine we are going to propose in this paper executes workflows represented in term of Petri Nets. Petri Nets have been chosen as *internal* model because of their formal semantics. The structure of a Petri-Net-model is formally defined by a set of *places*, a set of *transitions* and a set of *arcs* connecting places to transitions and vice versa (but not place to place or transition to transition). The *High Level Petri Nets* (HLPN) model extends classical Petri Nets with features that make them more suitable for workflow representation; an introduction to theoretical aspects of HLPN can be found in [3](in our paper we always refer to HLPN when the Petri Net term is used). The dynamic behavior of the net is described by using *tokens* which are associated to places; tokens enable transitions, make them *fire* and as a consequence they flow through the network. From a workflow perspective, a transition is associated to a task execution (job submission) and a token represents data that flows between tasks. An outline of a Petri-Net-based workflow engine can be described using the state chart diagram shown in Fig. 2. The engine needs to select enabled transitions, submits relative tasks to the Grid and monitors their execution; when a task ends the net state is updated, data (tokens) are moved and new transitions are selected to fire. The workflow execution continues until all the submitted jobs terminate and there is no further enabled transition. Unlike a formal Petri Net model, where transitions are *atomic* operations, we have also to deal with a transition *failure* (referred to a task failure). In scientific workflow, tasks are tipically operations which take raw data in input and produce refined data as output. As far as tasks are usually *idempotent*, the common failure recovery strategy (which is also used by DAGMan) consists in a task re-submission. However, in ordert to achieve at business workflows compatibility, we have to deal with different failure recovery strategies (i.e. rollback, choose an alternative task). This is made possible by pushing out of the engine the failure management: when a failure is recognized, it is handled by the workflow itself as shown in Fig. 2. Failure management could be explicity done by the user during the process design; or as a result of a workflow refinement discussed in [4].

The top layer aims at language *independence*. The basic idea is to make the workflow engine compatible with a large set of workflow languages. A pluggable system of parsers provides support for several languages in order to allow *collaboration* between WfMSs and support for *legacy* workflows. As we will see in more detail in the next section this layer has the responsibility to extrapolate the semantics behind a workflow description and translate it in

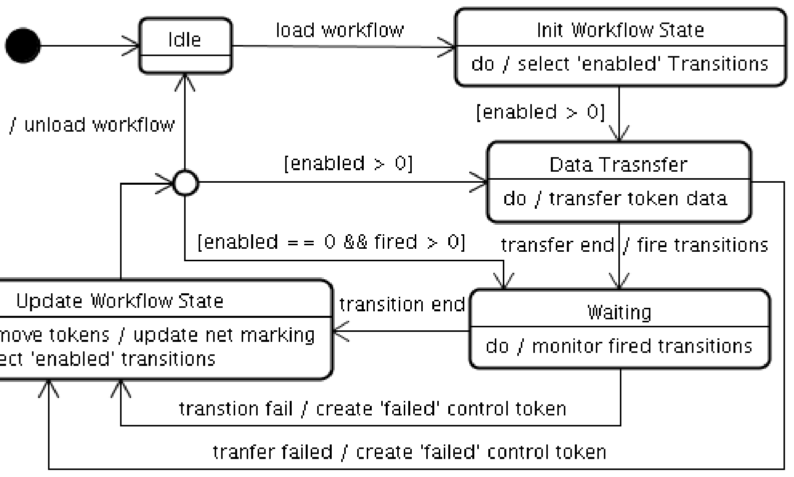
Idle
load workflow
Init Workflow State
do / select 'enabled' Transitions
[enabled > 0]
/ unload workflow
[enabled > 0]
Data Trasnsfer
do / transfer token data
[enabled == 0 && fired > 0]
transfer end / fire transitions
Update Workflow State
entry / move tokens / update net marking
do / select 'enabled' transitions
transition end
Waiting
do / monitor fired transitions
transtion fail / create 'failed' control token
tranfer failed / create 'failed' control token

Figure 2. The Workflow engine behavior

terms of a Petri Net. On the other side the gateway makes it possible to transfer parts of a process to a different WfMS.

Our interest is to build such WfMS in term of a *micro-kernel* pattern as proposed in [5]. A micro-kernel pattern [6]aims at the separation of a minimal functional core such as job submission, monitoring and data movement from extended functionality providing *extensibility*. That means advanced workflow features such as planning, scheduling strategies and QoS management should be built on top of the kernel API. A micro-kernel pattern provides *modularity* and *extensibility* which are fundamental properties for systems like a WfMS, where a standard is not well defined yet and they must be able to adapt easily to changing requirements.

3. Workflow Interoperability

As shown in Fig. 1 there are two kinds of interaction we would like to investigate: the first one (*A*) is about translation between different workflow description languages; the second one (*B*) is about synchronization between different workflow engines. Both aspects are part of the interoperability interface described in the WfMC's reference model [7].

As previously said many workflow languages exist due to the lack of a strong standard. However, *translation* from one language to another is often possible; what is needed is a language *parser*, a *model translator* and a *compiler*, as shown in the top layer of Fig. 1. Parsers have the responsibility to extract the workflow semantics from a description, expressed using a workflow language. As explained before, workflows can be described in terms of DAGs, Petri Nets, π-Calculus or Activity Diagrams, albeit with different expressivity levels (e.g. a DAG cannot describes an iteration of tasks). Conversion between these formalisms is provided by the model converter which represents the critical part of such process. In fact is not always possible to rapresent one model in terms of a different one; for example a Petri Net cannot always be converted in term of a DAG. Finally compilers translate the model into a specific (usually different from the initial) language description.

A debate exists around the best formalism to use for workflow description; Petri Nets and π-Calculus are widely used for workflow modeling. As far as both the formalisms are Turing-complete, the choice relies on the way these models deal with *workflow patterns*. Workflow patterns are a collection of well-known problems, and solutions, related to the support of process-oriented applications [8]. According to [9], Petri Nets outperform other formalisms in workflow description thanks to their formal semantics; also several *analysis techniques* exist in order to determine the properties (*correctness*, *deadlocks* and *boundary*) of a process design. As previously said, Petri Nets provide mechanisms for model conversion. DAGs can be simply represented in terms of Petri Nets; also π-Calculus based models (such as BPEL) can be translated in terms of Petri Nets; the semantics of such translation is discussed in [11]. This choice is also compatible with the one done by other CoreGRID partners like the Fraunhofer FIRST which introduced an XML based language called GWorkflowDL [10]that allows the representation of *abstract* and *concrete* workflows using Petri Nets. Fraunhofer FIRST also works, since several years, on a workflow enactment engine called GWES in which we would like to contribute with our work.

Compilers come into stage when a workflow model, or a part of it, needs to be represented using a specific language. For example a part of a process, usually a sub-workflow, can be transferred to a different WfMS; the internal Petri Nets representation must be converted in a language description the third-party WfMS understands. Unfortunately this kind of conversion is not always possible, for example there is no explicit semantics for translating a Petri Net in terms of π-Calculus formalism. A set of specific language compilers are needed in order to achieve at compatibility with *legacy* WfMSs.

Interoperability as described in the WfMC's reference model needs an engine level synchronization mechanism identified with letter *B* in Fig. 1. A runtime support for the interchange of various types of control information and transfer of workflow relevant and/or application data between different WfMSs. Synchronization can be useful even when several instances of the same workflow engine want to collaborate; it can be useful when we want to use the WfMS as *distributed* service to increase its performances.

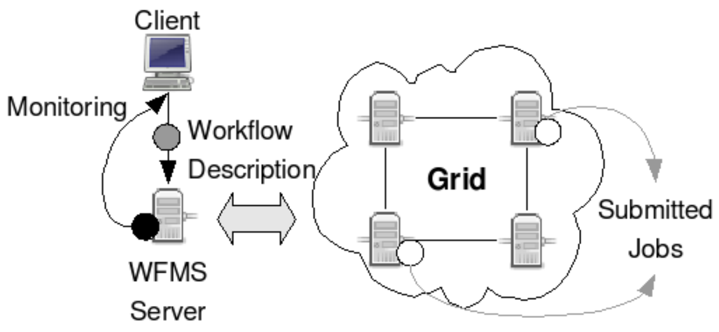

Figure 3. The WfMS deploy scenario 1

4. Implementation

The implementation will rely on the gLite middleware developed within the EGEE project, due to its maturity and its large adoption. Job submission in gLite is done by the Workload Management System (WMS) [12]; it comprises a set of Grid middleware components responsible for the distribution and management of tasks across Grid resources. The core component of the WMS is the Workload Manager (WM) whose purpose is to accept and satisfy requests for job management, expressed via a ClassAd-based Job Description Language (JDL). The WMS also supptorts the execution of single workflows expressed as DAGs. Job monitoring in the WMS is provided by the Job Logging and Bookkeeping Service (LB).

Many scientific workflows are expressed as JDL DAGs. As first step, our purpose is to provide a mechanism that allows those legacy processes to take advantage of the WfMS. Thanks to a JDL *parser* the DAG model can be extracted and then converted (using the *model converter*) into a Petri Net the WfMS can execute. Subsequently we would like also to investigate the integration of the BPEL workflow language providing a π-Calculus to Petri Net model translator as proposed in [11].

Initially the WfMS will run as a separate process on a dedicated server, as shown in Fig. 3. In this scenario the client sends the workflow description to the WfMS server, which executes it submitting jobs to the the Grid. The WfMS server also allows the client to monitor the workflow execution. This kind of solution is simple to realize but has some disadvantages: as far as a workflow could run for several days a failure in the WfMS server could cause the loss of the entire process and data. To avoid that the server must provide high *reliability* and recovery tools.

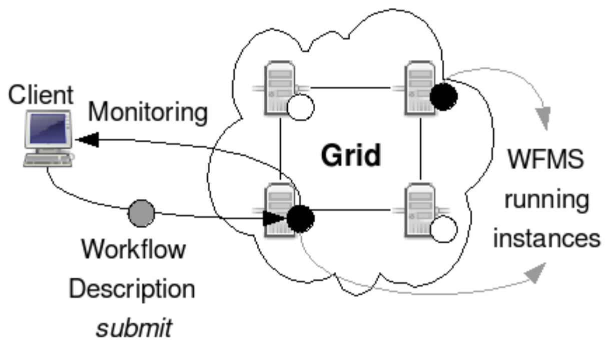

Figure 4. The WfMS deploy scenario 2

Later we will investigate an alternative solution that takes effort of the Grid environment to run users' workflows. In fact the Grid provides computational resources and mechanisms that helps failover management which makes run of the WfMS as a Grid job a strategic choice. Grid is a batch system where users send jobs and waits for their termination; a WfMS is also a sort of *complex job* which can be submitted to the Grid within the workflow description. As shown in Fig. 4, the client submits a workflow using the Grid middleware; a new WfMS instance will start on a Grid node (selected by the WMS) and it will use the Grid Abstraction Layer for job submission and monitoring.

As a consequence of these considerations, the WfMS must easily adapt to environment changes. Therefore during the development stage we will not focus on the system itself but in defining a set of basic functions, according to the micro-kernel pattern, which the WfMS will rely on, making changes simpler.

5. Conclusions and Future Work

In this paper we introduced a WfMS architecture with the intent to be compatible with the roadmap of the CoreGRID project. The main focus will be on language conversion and interoperability between WfMSs using language and model translators; Grid middleware independence will be satisfied thanks to a layered architecture. The combination of these features makes it possible for users to run their legacy workflows (usually written in different languages) on a large set of computational resources. In fact, coordination of workflows among several heterogeneous WfMSs is one of the main challenges in today WfMS research.

The WfMS we are proposing is quite simple compared to other WfMSs like Triana or GWES; for example it lacks QoS management, advanced planning techniques and so on. However, one of the purposes of this work is to introduce a generic *lightweight* WfMS core with basic functions where advanced functionalities can be easily and *dynamically* integrated thanks to the micro-kernel architecture. A WfMS for researchers who want to investigate high level aspects related to workflows management without taking care of low level problems such us job submission, data transfer and so on.

References

[1] Condor DAGMan, http://www.cs.wisc.edu/condor/dagman/

[2] D. Colling et al.: *Adding Instruments and Workflow Support to Existing Grid Architectures*, International Conference on Computational Science (3), 2006.

[3] Kurt Jensen: *An Introduction to the Theoretical Aspects of Colored Petri Nets*, Lecture Notes in Computer Science (Springer), 1994.

[4] A. Hoheisel and U. Der: *Dynamic workflows for Grid applications*, in: Proceedings of the Cracow Grid Workshop '03 (Cracow, Poland, 2003).

[5] Dragos A. Manolescu: *An extensible Workflow Architecture with Object and Patterns*, TOOLSEE 2001.

[6] Douglas Schmidt et al.: *Pattern-Oriented Software Architecture*, Siemens AG, pages 171-192, 2000.

[7] D. Hollingsworth: *Workflow management coalition: The workflow reference model*, Document TC00-1003, Workow Management Coalition, 1994.

[8] Workflow Patterns, http://www.workflowpatterns.com.

[9] W.M.P. van der Aalst: *The Application of Petri Nets to Workflow Management*, The Journal of Circuits, Systems and Computers, pages 21-66, 1998.

[10] Martina Alt et al.: *A Grid Workflow Language Using High-Level Petri Nets*, Lecture Notes in Computer Science (Springer), CoreGRID Technical Report Number TR-0032, 2006.

[11] Christian Stahl: *A Petri Net Semantics for BPEL*, Humboldt University Berlin, 2004.

[12] P. Andreetto et al.: *Practical Approaches to Grid Workload and Resource Management in the EGEE Project*, Conference for Computing in High-Energy and Nuclear Physics (CHEP 04), Interlaken, Switzerland, 27 Sept - 1 Oct 2004.