Workflow Management in the CrossGrid Project[1]

Anna Morajko[2], Enol Fernández[2], Alvaro Fernández[3], Elisa Heymann[2], Miquel Ángel Senar[2]

[2]Universitat Autònoma de Barcelona, Barcelona, Spain
{ania,enol}@aomail.uab.es;{elisa.heymann,miquelangel.senar}@uab.es
[3]Instituto de Física Corpuscular, Valencia, Spain
alvaro.fernandez@ific.uv.es

Abstract. Grid systems offer high computing capabilities that are used in many scientific research fields and thus many applications are submitted to these powerful systems. Parallel applications and applications consisting of inter-dependent jobs may especially be characterized by a complex workflow. Therefore, Grid systems should be capable of executing and controlling workflow computations. This document sets out our approach to workflow management in a Grid environment. It introduces common steps on how to map an application workflow to the DAG structure, and how to carry out its execution and control. We present the Workflow Management Service (WFMS) implemented and integrated as a part of the CrossGrid project. The purpose of this service is to schedule workflow computations according to user-defined requirements, also providing a set of mechanisms to deal with failures in Grid.

1 Introduction

The Grid represents distributed and heterogeneous systems and involves coordinating and sharing computing, application, data, storage, or network resources across dynamic and geographically dispersed organizations [1]. Grid systems offer high computing capabilities that are used in many scientific research fields. They facilitate the determination of the human genome, computing atomic interactions or simulating the evolution of the universe. Many researchers have therefore become intensive users of applications with high performance computing characteristics. There are projects such as GriPhyn [2], DataGrid [3], GridLab [4] or Crossgrid [5] that provide the middleware infrastructure to simplify application deployment on computational grids.

The main objective of the CrossGrid project is to incorporate a collection of machines distributed across Europe, and to provide support especially for parallel and interactive compute- and data-intensive applications. As a result of this project, parallel applications compiled with the MPICH library (and using either ch-p4 [6] or Globus2 [7] device) are executed on Grid resources in a transparent and automatic way. The workload management system that we have implemented as part of the

[1] This work has been partially supported by the European Union through the IST-2001-32243 project "CrossGrid" and partially supported by the *Comisión Interministerial de Ciencia y Tecnología* (CICYT) under contract TIC2001-2592.

CrossGrid middleware carries out all necessary steps incurred from the time that the application is submitted by the user until the end of its execution (i.e. potential resource discovery, selection of the best matched resources and execution tracking). Our workload management system has been designed to manage Grid-specific details of the application execution with minimal effort by the user.

In the previous work [8], we described the specific details related to the execution of MPI applications on the Grid. In this paper, we focus on the additional service that we have included to support workflow computations. Many applications may consist of inter-dependent jobs, where information or tasks are passed from one job to another for action, according to a set of rules. Such applications known as workflows consists of a collection of jobs that need to be executed in a partial order determined by control and data dependencies. Workflows are an important class of applications that can take advantages of the resource power available in Grid infrastructures, as has been shown in the LIGO [9] pulsar search, several image processing applications [10] or physics experiment ATLAS [11]. The execution of such an application may be very difficult. Normally, a user should submit to a Grid system manually, job by job, following the rules of dependencies that appear between jobs. The manual tracking of the application workflow may be very ineffective, time consuming and may produce many errors in application execution. Therefore, we present a solution to the automatic management of the application workflows applied in the CrossGrid project.

Section 2 briefly presents related work. Section 3 introduces a general overview of the workload management in the CrossGrid, indicating new features for workflow support. Section 4 sets out the workflow notation and Section 5 introduces details of the workflow management. Section 6 shows the results of the probes conducted using a workflow whose structure is representative of the ATLAS experiment. Finally, Section 7 presents the conclusions to this study.

2 Related work

A number of studies in Grid systems provide general-purpose workflow management. The Condor's DAGMan [12] – DAGMan (Directed Acyclic Graph Manager) is a meta-scheduler for Condor. DAGMan manages dependencies between jobs, submitting these to Condor according to the order represented by a DAG and processes the results. The DAG must be described in an input file processed by the DAGMan and each node (program) in the DAG needs its own Condor submit description file. DAGMan is responsible for scheduling, recovery, and reporting for the set of programs submitted to Condor. This scheduler focuses on the execution of workflows in a local cluster managed by the Condor system.

Pegasus system [13] – Planning for Execution in Grids – was developed as part of the GriPhyN project. Pegasus can map scientific workflows onto the Grid. It has been integrated with the GriPhyN Chimera system. Chimera generates an abstract workflow (AW), Pegasus then receives such a description and produces a concrete workflow (CW), which specifies the location of the data and the execution platforms. Finally, Pegasus submits CW to Condor's DAGMan for execution. This system focuses on the concept of virtual data and workflow reduction. Triana [14] is a

Problem Solving Environment (PSE) that provides a graphical user interface to compose scientific applications. A component in Triana is the smallest unit of execution written as Java class. Each component has a definition encoded in XML. Such created application's graph can then be executed over Grid network using the GAP interface. Unicore [15] stands for Uniform Interface to Computing Resources and allows users to create and manage batch jobs that can be executed on different systems and different UNICORE sites. The user creates an abstract representation of the job group (AJO – Abstract Job Object) that is then serialized as a Java object, and in XML format. UNICORE supports dependencies inside the job group and ensures the correct order of the execution. Its job model can be described as directed acyclic graphs. UNICORE maps the user request to system specification, providing job control. In contrast to our work, Triana, Pegasus and Unicore lack resource brokerage and scheduling strategies.

GridFlow [16] supports a workflow management system for grid computing. It includes a user portal in addition to services of global grid workflow management and local grid sub-workflow scheduling. At the global level, the GridFlow project provides execution and monitoring functionalities. It also manages the workflow simulation that takes place before the workflow is actually executed. This approach is applicable only in the case of having performance information about job execution. At the local grid sub-workflow level, scheduling service is supported.

3 CrossGrid workload management

This section presents the main components that constitute the Workload Management System (WMS) applied in the CrossGrid project. A user submits a job to a Scheduling Agent (SA) through a Migrating Desktop or command line (see Figure 1). The job is described by a JobAd (Job Advertisement) using the EU-Datagrid Job Description Language (JDL) [17], which has been extended with additional attributes to support interactive and parallel applications, as well as workflows.

To support the workflow execution, we have included specific service into the WMS. Workflows have a special treatment at the beginning as they are passed from the SA to the Condor's DAGMan, which is a specialized scheduler module that submits each individual job to the SA when job dependencies have been satisfied.

For each simple job (submitted directly by the user or by the Condor's DAGMan), the SA follows the same steps. It asks the Resource Searcher (RS) for resources to run the application. The main duty of the RS is to perform the matchmaking between job needs and available resources. The RS receives a job description as input, and, as output, returns a list of possible resources within which to execute the job. Computing resources are available as Computing Elements (CE), which provide the abstraction of a local farm of Working Nodes (WN). This local farm (or CE) is accessed through a Gatekeeper. The list of resources returned by the Resource Searcher consists of a Computing Elements list. Subsequently, the Scheduling Agent selects a CE on which to run the job. The SA passes the job and the first-selected CE to the Application Launcher (AL), who is responsible for the submission of that job on the specified CE. The AL passes the job to Condor_G [18], which manages a queue of jobs and

resources from sites where the job can be executed. Due to the dynamic nature of the Grid, the job submission may fail on that particular CE. Therefore, the Scheduling Agent will try the other CEs from the list returned by the Resource Searcher. Finally, the Scheduling Agent notifies the user of the result.

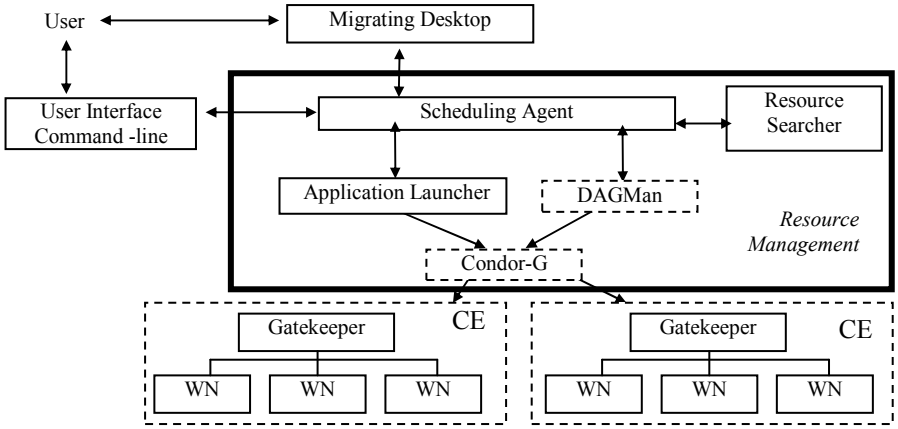

Fig. 1. Architecture of the Workload Management System.

4 Workflow notation and specification

There are many complex applications that consist of inter-dependent jobs that cooperate to solve a particular problem. The completion of a particular job is the condition needed to start the execution of jobs that depend upon it. This kind of application workflow may be represented in the form of a DAG – a directed acyclic graph. A DAG is a graph with one-way edges that does not contain cycles. It can be used to represent a set of programs where the input, output, or execution of one or more programs is dependent on one or more other programs. The programs are nodes (vertices) in the graph, and the edges (arcs) identify the dependencies of these programs. Figure 2 presents an example DAG that consists of 4 nodes lying on 3 levels. The execution of the indicated DAG consists of three successive steps:

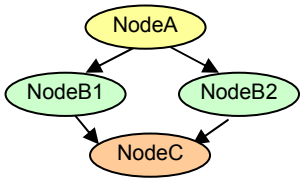

Fig. 2. Example DAG.

1. Execution of the node NodeA from the first level.
2. Parallel execution of nodes NodeB1 and NodeB2 from the second level. The execution can start if and only if the execution of the node NodeA is successful.

3. Execution of the node NodeC from the third level. The execution can start if and only if the execution of all nodes from the level two is successful.

4.1 Workflow specification using JDL

Workflows, similar to a normal job, are specified in a text file using DataGrid JDL format extended appropriately to support DAG structure and control. The workflow description has two components: specification of dependencies between computations (node dependencies) and specification of computation (node description). Below, we present an example JDL file for a workflow specified as in Figure 2.

```
[
type = "dag";
nodes = [
      dependencies={{NodeA,{NodeB1,NodeB2}},{{NodeB1,NodeB2},NodeC}};
      NodeA = [
              node_type = "edg-jdl";
              description = [
                     Executable = "jobA.sh";
                     InputSandbox = {" jobA.sh"};
              ];
      ];
      NodeB1 = [
              node_type = "edg-jdl";
              node_retry_count = 3;
              app_exit_code = { 10, 11 };
              file = "jobB1.jdl";
      ];
      NodeB2 = [
              node_type = "edg-jdl";
              file = "jobB2.jdl";
      ];
      NodeC = [
              node_type = "edg-jdl";
              file = "jobC.jdl";
      ];
];
]
```

Job specification

Node specification

4.2 Dependencies between nodes

Each dependence is specified as a pair of elements positioned between brackets, where the meaning is that the second element depends on the first. Both elements may be formed by a set of elements written between brackets. This indicates a dependence of many-to-one, one-to many or many-to-many elements. Therefore, considering the example DAG, there are few possibilities to specify the attribute *dependencies*:

- `{{NodeA,NodeB1},{NodeA,NodeB2},{NodeB1,NodeC},{NodeB2,NodeC}}`
- `{{NodeA,NodeB1},{NodeA,NodeB2},{{NodeB1,NodeB2},NodeC}}`
- `{{NodeA,{NodeB1,NodeB2}},{{NodeB1,NodeB2},NodeC}}`

4.3 Node description

The attribute `nodes` contains the list of nodes that form the DAG. Each node represents a job to be executed and contains node-specific attributes, as well as the job specification. The attributes for a node description are:

- `node_type` – specifying a type of a node. This attribute is mandatory and must contain the value "edg-jdl", as the node is a normal job written in JDL format.
- `node_retry_count` – specifying how many times a node execution may be retried in the case of failure. This attribute is optional. If the user specifies this attribute for a node; hence, if it fails, this particular node will be automatically retried. Otherwise, this attribute will be set to the default value.
- `app_exit_code` – specifying the possible exit codes for a job. If a node fails because of the application failure (e.g. segmentation fault, division by 0, a file already registered in a Storage Element), then a job should be aborted. However, when a job fails because given resources fail (e.g. a machine failure, Condor/PBS queue problems), it should be automatically retried. By default, in both cases, the node will be retried until `node_retry_count`. The attribute `app_exit_code` provides the possibility to control the job exit code and terminate job execution in case of failure. If this attribute contains certain values, this means that the job may return such values and that they are recognized by the user. If the job execution returns one of the specified values, the node will not be retried. Otherwise, the job will be retried automatically according to `node_retry_count` (if the maximum of retries is not reached) and submitted to other CE.
- `description / file` – specification of the job; A job can be a normal, single job or an MPI job. There are two ways to specify a job:
  - Via an attribute called `description`, where a job is specified directly inside this attribute in JDL:

```
description = [
        Executable = "jobA.sh";
        InputSandbox = {" jobA.sh"};
        …
];
```

  - Via an attribute `file`, where a job is specified in the indicated JDL file

```
file = "jobA.jdl";
```

5 Scheduling application workflow

To support application workflows, we have extended the Scheduling Agent with a new component called Workflow Manager Service (WFMS). The WFMS is executed within the Resource Broker machine that contains the WMS presented in Section 3. Specification and implementation of the workflow supported in the CrossGrid project takes advantages and leverages of a preliminary work presented in [19]. As shown in Figure 1, workflows are passed from the Scheduler Agent to the Condor's DAGMan. DAGMan is an iterator on the DAG, whose main purpose is to navigate through the graph, determine which nodes are free of dependencies, and follow the execution of corresponding jobs, submitting these to the SA. While DAGMan provides us with the

automatic graph management, the WFMS is responsible for searching grid resources, controlling errors and retrying the execution of failed nodes avoiding CEs' repetition. A DAGMan process is started for each workflow submitted to the WMS. If there is more than one DAG to execute, a separate DAGMan process will be started for each DAG. A set of steps must be performed for each node in the workflow:

1. Initial phase – preparing all necessary information for the node execution. A suitable resource to run the job is searched by the Resource Searcher and the job is then passed to Condor-G, which will be responsible for submitting this to the remote site. If no resources are found, the WFMS will mark the node as failed, which implies the end of the node execution. This node can be automatically retried according to the `node_retry_count` value.
2. Job execution on the remote site.
3. Final phase – checking the job execution return code. If the job was executed successfully, it is marked as Done. Otherwise, the WFMS compares the return value to the attribute `app_exit_code`; if the return value is one of the values specified by the user, the job is not retried; in any other, case the job is marked as failed and is retried according to the `node_retry_count` value.

The WMS also supports the functionality by which to submit a failed workflow and execute only those nodes that have not yet been successfully executed. This workflow is automatically produced by the WMS when one or more nodes in the workflow has resulted in failure, making the application execution impossible to finish. If any node in the a workflow fails, the remainder of the DAG is continued until no more forward progress can be made, due to workflow's dependencies. At this point, the WFMS produces a file called a Rescue DAG, which is given back to the user. Such a DAG is the same as the original workflow file, but is annotated with an indication of successfully completed nodes using the `status=done` attribute. If the Rescue DAG is resubmitted using this Rescue DAG input file, the nodes marked as completed will not be re-executed.

A DAG is considered as a normal, single job unit of work. A DAG execution goes through a number of states during its lifetime:

- *Submitted* – The user has submitted the DAG using User Interface but it has not yet been processed by the Network Server
- *Waiting* – The DAG has been accepted by the Network Server
- *Ready* – The DAG has been processed by the Workload Manager that has decided to run a Condor's DAGMan
- *Running* – The DAGMan process is running; the DAG is passed to the DAGMan
- *Done* – The DAG execution has finished
- *Aborted* – The DAG execution has been aborted because of an external reason
- *Cancelled* – The DAG execution has been cancelled by the user
- *Cleared* – The Output Sandbox of all nodes has been transferred to the UI

5.1 User-level commands for workflow management

A set of user commands is available, allowing users to submit and control the application workflow execution. The list of commands is as follows:

- edg-dag-submit – this command submits a DAG

- edg-job-status – this command checks the status of the submitted DAG
- edg-job-cancel – this permits a user to cancel the execution of a DAG
- edg-job-get-output – this command obtains output for all jobs of the DAG
- edg-job-get-logging-info – this presents logging info for a DAGMan execution

6 Experimental results

We conducted a number of experiments on parallel/distributed applications to study how this approach works in practice. We present a DAG whose structure is representative of the ATLAS (Atlas A Toroidal LHC ApparatuS) experiment [9] – the largest collaborative effort in the physical sciences. This experiment contains the following successive steps: event generation (evgen), simulation (sim), digitalization (digi). The first process does not take any input data, but rather generates certain output data. The second process processes data generated by the previous step, giving another data as result. The third process repeats the scheme of the sim step. To provide an efficient execution of the experiment, each step can be divided into N parts, where each part processes a subset of data as it is shown in Figure 3 (because of limits on space, we do not present the JDL specification of this DAG):

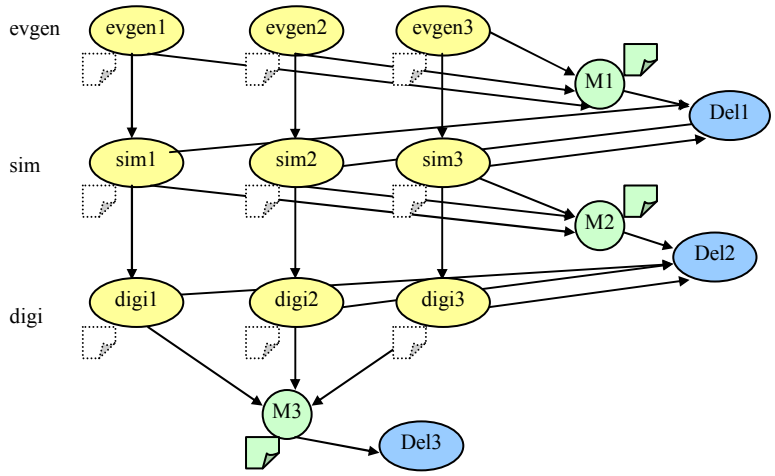

Fig. 3. An example DAG that represents the workflow of the ATLAS experiment.

- evgen – this contains N sub-nodes, where each sub-node performs the partial event generation, and the generated partial result file is copied and registered in the Crossgrid storage using EDG Replica Manager [20].
- sim – this contains N sub-nodes, where each one copies the partial file generated by an evgen sub-node; it then performs the simulation on the partial data and generates a partial file. Finally, it copies and registers this file in the storage.
- digi – this contains N sub-nodes, where each one copies the partial file generated by a sim sub-node; it then performs digitalization on the partial data and generates a partial file. Finally, it copies and registers this file in the storage.

There is, additionally, a need for further steps that merge the temporal files generated by all parts of the considered step and remove such temporal files:

- Merge – this copies the files generated by all nodes (separately for each level) and merges them; it saves the results to a file and finally copies and registers this file in the storage; these merged files will be the result of the experiment.
- Delete – this deletes the files generated by all levels except Merge; this step can be carried out separately for each level.

Figure 4 presents an example execution of the workflow in the ATLAS experiment. The horizontal axis represents time, the vertical shows Computing Elements in which the workflow may be executed (Poland, Spain or Portugal). In this example, all nodes from the Event Generation level and merge operation have been successfully finished. During the execution of the simulation nodes, one of these has failed (sim2). This node has been retried automatically, but has failed again. It has not been possible to retry it as, e.g., a user specified `node_retry_count=1`. The workflow execution has terminated as failed. However, the Rescue DAG, which contains nodes annotated as already done, is provided for the user, who may therefore submit the failed workflow executing only those nodes that have not been successfully finished.

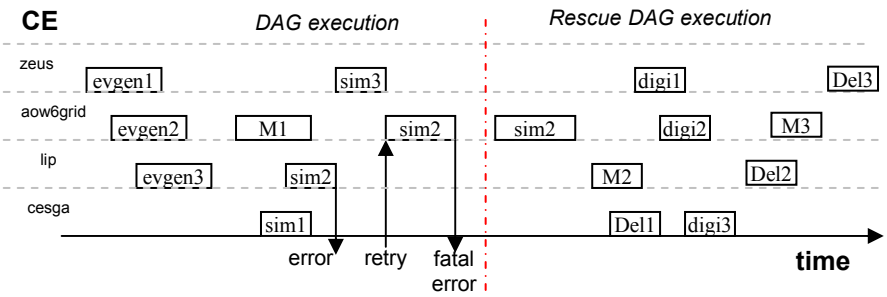

Fig. 4. An example execution of the DAG for the ATLAS experiment.

6 Conclusions

The Grid system offers high computing capabilities for users in many scientific research fields. A great number of applications are made for a set of jobs that depend on each other. Generally, the execution of such applications in the grid environments requires users' intervention and the manual execution of each job, step by step, simulating their dependencies. It is therefore necessary to provide a good, reliable and simple service which automatically carries out the task of mapping the application workflow to the Grid.

In this paper, we have presented the solution to this problem, which has been applied within the EU-Crossgrid project. To provide workflow execution, we have represented such a workflow in the form of DAGs. The DAG execution is provided by the DAG Manager service and is integrated with the CrossGrid's Workload

Management System (WMS). Our implementation is based on the DAGMan scheduler provided by the Condor group and is targeted to LCG-2 middleware.

Many aspects could be introduced to improve DAG support. For example, support for DAG in DAG (the possibility of specifying a node as another DAG) or the integration of DAG into the Migrating Desktop and Web Portal that provides a user-friendly interface by which to interact with the Grid.

7 References

1. I. Foster, C. Kesselman (Editors), "The GRID Blueprint for a New Computing Infrastructure". Morgan Kauffmann Publishers. 1999.
2. GriPhyN: The Grid Physics Network. http://www.griphyn.org
3. The DataGrid Project. http://www.eu-datagrid.org
4. GridLab: A Grid Application Toolkit and Testbed: http://www.gridlab.org
5. The EU-Crossgrid project. http://www.eu-crossgrid.org
6. W. Gropp, E. Lusk, N. Doss, A. Skjellum, "A high-performance, portable implementation of the MPI message passing interface standard". Parallel Computing, 22(6), pp.789-828. 1996.
7. N. Karonis, B. Toonen, I. Foster, "MPICH-G2: A Gridenabled implementation of the message passing interface". Journal of Parallel and Distributed Computing, to appear. 2003.
8. E. Heymann, M.A. Senar, A. Fernandez, J. Salt, "The Eu-Crossgrid approach for Grid Application Scheduling". 1st European Across Grids Conference, LNCS series, pp.17-24. February, 2003.
9. B. Barish, R. Weiss, "Ligo and detection of gravitational waves". Physics Today, 52 (10). 1999.
10. S. Hastings, T. Kurc, S. Langella, U. Catalyurek, T. Pan, J. Saltz, "Image processing on the Grid: a toolkit or building grid-enabled image processing applications". In 3rd International Symposium on Cluster Computing and the Grid. 2003.
11. http://atlasexperiment.org/
12. http://www.cs.wisc.edu/condor/dagman/
13. E. Deelman, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, S. Patil, M.H. Su, K. Vahi, M. Livny, "Pegasus : Mapping Scientific Workflows onto the Grid". Across Grids Conference. Nicosia, Cyprus, 2004.
14. M. Shields, "Programming Scientific and Distributed Workflow with Triana Services". GGF10 Workflow Workshop. Berlin, March, 2004.
15. UNICORE Plus - Final Report (2003). http://www.unicore.org
16. J. Cao, S.A. Jarvis, S. Saini, G.R. Nudd, "GridFlow: Workflow Management for Grid Computing". 3rd International Symposium on CCGrid. Japan, May 2003.
17. F. Pazini, "JDL Attributes - DataGrid-01-NOT-0101-0_4.pdf" http://www.infn.it/workload-grid/docs/DataGrid-01-NOT-0101-0_4-Note.pdf. December, 2001.
18. James Frey, Todd Tannenbaum, Ian Foster, Miron Livny, and Steven Tuecke, "Condor-G: A Computation Management Agent for Multi-Institutional Grids", Journal of Cluster Computing, vol. 5, pages 237-246, 2002.
19. Data grid: Definition of the architecture, technical plan and evaluation criteria for the resource coallocation framework and mechanisms for parallel job partitioning. WP1: Workload Management. DataGrid-01-D1.4-0127-1_0. Deliverable DataGrid-D1.4. 2002.
20. L. Guy, P. Kunszt, E. Laure, H. Stockinger, K. Stockinger, "Replica Management in Data Grids". Technical report, GGF5 Working Draft. July 2002.