A Workflow Management Platform for Scientific Applications in Grid Environments

Alexandru Costan, Florin Pop, Ciprian Dobre, Valentin Cristea
University Politehnica of Bucharest
Computer Science Department
Bucharest, Romania
Email: {alexandru.costan, florin.pop, ciprian.dobre, valentin.cristea}@cs.pub.ro

***Abstract*—Workflow management systems allow the development of complex applications at a higher level, by orchestrating functional components without handling the implementation details. Although a wide range of workflow engines are developed in enterprise environments, the open source engines available for scientific applications lack some functionalities or are too difficult to use for non-specialists. Moreover, the key middleware services in grids, like meta-schedulers, employ complex resource matching logic, but lack control flow orchestration capabilities. On the other hand, workflow engines suitably control process logic but are unaware of execution requirements of tasks. In this paper we present PEGAF, a workflow management platform for distributed systems targeted at scientific applications. Our platform provides features like an intuitive way to describe workflows and flexible fault tolerance support, while integrating both workflow orchestration and meta-scheduling. We validate our integrated platform within several test scenarios, outlining a significant performance increase compared to traditional workflow orchestration solutions.**

Keywords: workflow management, distributed systems, grids, scheduling, fault tolerance.

I. Introduction

Distributed applications, both in the academic and enterprise environments, are becoming more and more complex, requiring the orchestration of multiple services or programs into workflows. Workflow systems are built in order to assist the user in developing complex applications at a higher level, by organizing the components and specifying the dependencies among them. Nowadays, business workflow engines provide a wide range of features suitable for enterprise applications. However, for scientific applications, even though a number of open source workflow systems are available, many of them are too difficult to use for non-specialists (some of them lack a graphical interface), or are restricted to a specific type of application or on a single middleware platform. These problems have been impeding the adoption of workflow-based solutions in the scientific community.

In this paper we present a workflow management platform for distributed systems, targeted at scientific applications, providing solutions for several issues in large scale distributed environments. We aim at a flexible workflow structure, allowing the orchestration of services and also of plain executable programs (so that users be able to introduce legacy applications in their workflows). Our platform relies on efficient mechanisms for data handling, as scientific applications usually produce significant amounts of data; the mechanisms are based on the data replication services provided by the underlying middleware. We enforce comprehensive fault tolerance support, with configurable policies. As semantics and side effects vary from one application to another, we believe that the users should be able to select from multiple fault tolerance approaches the one that is the most suitable for a particular workflow. We augmented the platform with an intuitive way to specify workflows, based on ontologies specific to the application domains, allowing users to work with abstract components that hide the implementation details. A decentralized model of a global meta-scheduler is responsible for managing connections with clients and local meta-schedulers. In addition to this, the global meta-scheduler also manages the efficient mapping of the jobs with resources (Web Services).

The workflow management platform is based on three layers of main components. A high-level module provides a user interface for defining abstract workflows, by managing domain specific ontologies. The middle-level layer has the role of a workflow engine, orchestrates WS-BPEL based workflows and enforces the fault tolerance support. The low-level module is responsible for scheduling the workflow activities and services onto the distributed system's physical resources, relying upon the available middleware.

We have started by studying the facilities offered by the most commonly used workflow engines for scientific applications, from the point of view of the requirements presented above. Although some workflow engines provide advanced features for abstract workflows, data management or fault tolerance, they lack functionality in what concerns the other aspects. As a consequence, we consider the approach of starting from an existing open source workflow engine and implementing additional functions that are required for the purposes of our project. The engine we have studied is ActiveBPEL, one of the most widely used engines for WS-BPEL, and we introduce here an architectural model of the modified ActiveBPEL engine, augmented with a new set of modules that implement the additional functions.

The remainder of this paper is organized as follows. Section 2, presents a functional analysis of existing workflow engines and arguments our choice for ActiveBPEL. Section 3 introduces the systems design of our platform, while Section 4 details the interface for abstract workflows specification.

Section 5 presents the workflow engine with its support for failure handling and in Section 6 we survey the underlying scheduling module. An illustration is given in Section 7 while Section 8 presents the performance evaluation results. Section 9 concludes this paper.

II. Related Work

In order to choose our underlying workflow engine, we surveyed several existing solutions that are most frequently used in scientic applications. We were interested by several aspects: programming paradigm for the workflow language, the type of the orchestrated components (jobs or services), the standardization of the used language, existing support for data management and fault tolerance.

Condor DAGMan Stork [2] was developed as a a batch scheduler specialized in data placement and data movement which understands the semantics and characteristics of data placement tasks and implements techniques specific to queuing, scheduling, and optimization of these type of tasks. Stork acts like an I/O control system (IOCS) between the user applications and the underlying protocols and data storage servers. It provides complete modularity and extendibility. The users can add support for their favorite storage system, data transport protocol, or middleware very easily. If the transfer protocol specified in the job description file fails for some reason, Stork can automatically switch to any alternative protocols available between the same source and destination hosts and complete the transfer. Thus, Stork can interact with higher level planners and workflow managers. Stork applies some of the traditional job scheduling techniques common in computational job scheduling to the data placement jobs: First Come First Served, Shortest Job First, Multilevel Queue Priority, Random Scheduling and Auxiliary Scheduling of Data Transfer Jobs. These techniques are applied to all data placement jobs regardless of the type. After this ordering, some job types require additional scheduling for further optimization.

Pegasus [3] enables scientists to construct workflows in abstract terms without worrying about the details of the underlying cyberinfrastructure or the particulars of the low-level specifications required by the cyberinfrastructure middleware. As part of the mapping, Pegasus automatically manages data generated during workflow execution by staging them out to user-specified locations, by registering them in data catalogs, and by capturing their provenance information. Since Pegasus dynamically discovers the available resources and their characteristics, and queries for the location of the data (potentially replicated in the environment), it improves the performance of applications through: data reuse to avoid duplicate computations and to provide reliability, workflow restructuring to improve resource allocation, and automated task and data transfer scheduling to improve overall workflow runtime. Pegasus also provides reliability through dynamic workflow remapping when failures during execution are detected. Currently, Pegasus schedules all the data movements in conjunction with computations. However, as the new data placement services are being deployed within the large-scale collaborations, workflow management systems such as Pegasus need to be able to interface and efficiently interact with the new capabilities.

Karajan is flexible in terms of interoperability by supporting the use of providers that allow middleware selection at runtime: GT2, GT3, GT4 or Condor [4], [11].

In Taverna [1] and ActiveBPEL workflows are seen as web services. The difficulty of implementation is hidden, users are presented a high-level interface. Interoperability for Taverna is limited to MyGrid, while ActiveBPEL can submit jobs to any middleware offering web services. Triana [6] is middleware agnostic: supports P2P, web services and Grids. Triana's API for accessing Grid services, is written in such a way that new modules can be added, to achieve interoperability with different middleware platforms. Triana jobs do not have web interfaces, communication is done only through the input/output files, and submission is performed by a resource manager (GRAM1 or GRMS2) [12].

We noticed a poor support for failure handling in most systems, usually consisting in stopping process execution and reporting the failure. However, manual resolution is not always applicable in large scale distributed environments; therefore automatic failure handling is needed. We conclude that improvements regarding the fault tolerant behavior that these systems provide are essential in order to make workflow systems more accessible to people from a multitude of scientific fields. On average, the fault tolerant performances of the above mentioned systems are acceptable from a common applications' point of view but they are unacceptable when it comes to long running, compute intensive applications. Until now efforts have been concentrated on correctly specifying and deploying such orchestration processes with the use of Web Services.

Many workflow engines work over a single type of middleware, besides those that enable web service orchestration (using WS-BPEL, for example) and should work with any middleware providing web services. This is another reason for choosing ActiveBPEL as our underlying engine for the proposed platform. The engine also comes with native failure handling and compensation support which facilitate checkpointing, essential for our targeted scientific applications. In addition, ActiveBPEL is based on a modular architecture which enables extensibility, is open source and well documented.

III. System Design

As we have shown in the previous section, although several open source workflow engines are available for executing scientific applications in distributed environments, most of them lack important features concerning fault tolerance, abstract workflows, data handling and user interface. We note however that some of the existing engines are based on highly expressive languages and provide advanced process management, transaction handling, database persistence and other mechanisms. As a consequence, we chose the solution

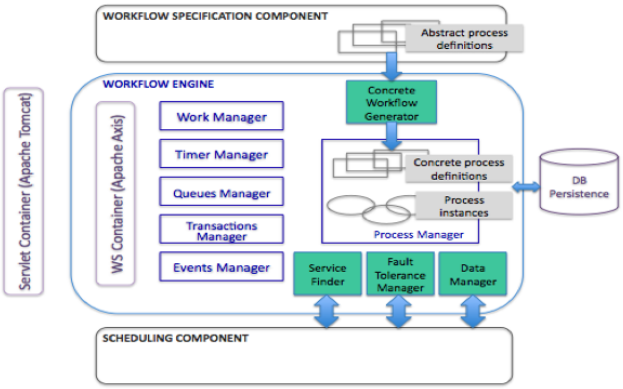

Fig. 1. The PEGAF platform architecture, based on ActiveBPEL

of starting from an open source workflow engine and building additional modules to satisfy our requirements.

The workflow engine we propose is ActiveBPEL, the most frequently used open source BPEL engine, integrated in several research projects. We briefly describe as follows the ActiveBPEL architecture and the extensions implemented for our project. ActiveBPEL runs on top of the Apache Tomcat servlet container, and uses an embedded version of Apache Axis for message communications. Figure 1 presents the main components of ActiveBPEL (in blue) and our proposed extensions (in green). Among the services used in ActiveBPEL for handling processes, which are named Managers, the most important one is the Process Manager. The Process Manager oversees the instantiation and execution of processes and activities. When a process is deployed, the engine analyzes the BPEL sources and generates an internal representation of the process; then, when the user requires the execution of the process, a new instance is created by the Process Manager. The Process Manager is also responsible with instantiating activities and associating them with states (inactive, executing, finished, faulted etc.) during their life cycle. The Queue Manager handles incoming messages and events addressed to the process activities, by building a queue with the activities that are waiting for messages. The Work Manager schedules asynchronous operations, based on "work objects" which are a specialized alternative to threads. We also mention the Time Manager, which provides support for timed operations (like suspending or waiting), and the Transaction Manager, which implements methods for working with transactions.

We introduced several new components in the ActiveBPEL engine: the Concrete Workflow Generator transforms abstract workflows into concrete workflows, the Service Finder maps service port types with sets of corresponding available services, the Data Manager implements efficient data handling mechanisms and the Fault Tolerance Manager, which applies the policies specified by the user for handling faults. The Service Finder component consists of work presented in Service Binder while the Data Management module implements the data placement algorithms. Due to space constraints, we do not details these components here, but refer the reader to the indicated papers. The scheduling component is represented by the global meta-scheduler. Is uses the Opportunistic Load Balancing Algorithm in order to assign jobs to local schedulers. Furthermore, the scheduler consider some fault-tolerance mechanism within the Grid environment. There are also possible to use batch scheduling algorithms. We detail these components in the following sections.

IV. The Abstract Workflows Handler

A good workflow specification and translation tool should be able to: represent abstract and concrete workflows, allowing different degrees of abstraction; provide means to express non functional requirements like adding semantics to both service description and workflow structure; allow handling dynamics; define parameters to describe Grid oriented services and workflows without dependencies on specific models infrastructure. In this section we present the Abstract Workflow Handler. This component manages all semantic aspects of the client framework providing tools and APIs for managing ontologies and their concepts. It enables users to access information dependent on the specific application domain they are interested in, to compose the workflow using the task templates available in the working domain or other user defined templates.

The process of generating a complete functional workflow is made up of three stages: the Service Pre-fetch Stage, the Service Generation Stage and the Workflow Generation Stage (Figure 2) mapped to the main building blocks of the semantic component: the Ontology Reader, the Service Builder and the three File Generators. During the first two stages, the data flow is sequential, as each functional block takes the raw data, performs the necessary operations and then passes it to the next block. In the last stage, the data flow becomes parallel, because at this moment, each component can be generated independently. We use an ontology written in OWL-S [5] to annotate existing Web services with semantic data. Basically, each Web service has an associated goal, representing the type of action it is able to perform. Every time a user inserts a new goal, its web service equivalent is searched within the ontology. When found, the data is parsed and the relevant information is stored in a Java object. However, there are cases when a goal is too complex to have only one associated web service. At this moment, it is recursively broken into simpler sub-goals until a Web service has been found for each generated sub-goal.

During the service pre-fetch stage, the Ontology Reader plays the role of a service analyzer. It extracts minimal information about the service, which is necessary in order to initialize data and then stores the number of user inputs and whether the goal provided can be directly satisfied or it has to be broken into several sub-goals. The output of this phase is always a list of services. If the Ontology Reader finds a service which is tagged as simple, it means there is a one-to-one relationship between the process (the workflow) and the service, or, in other words, the process is made up of a

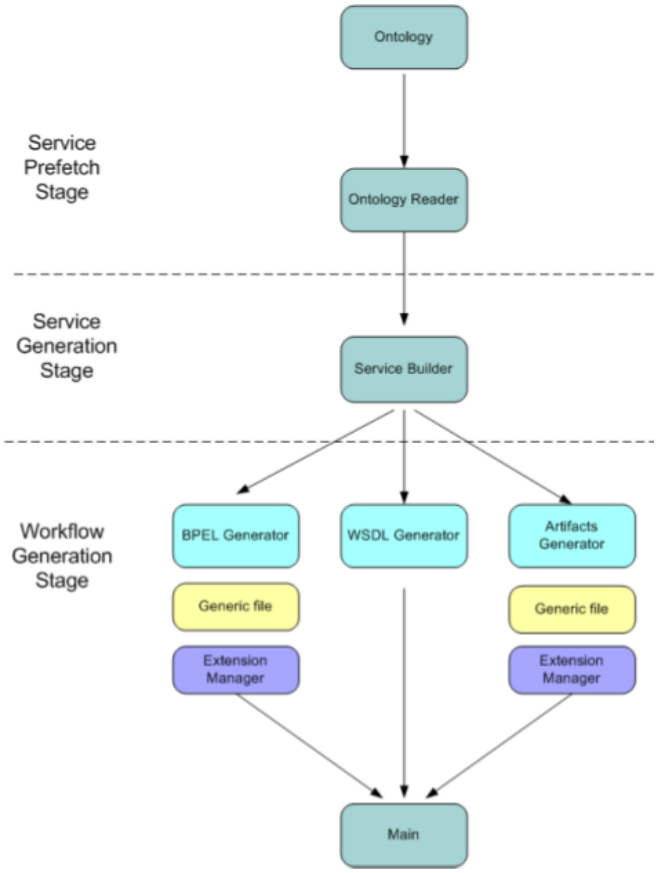

Fig. 2. The Concrete Workflow Generator modules

single Web service. In this case, the list built during the prefetch stage will contain a single element. Otherwise, if the Ontology Reader finds a service which is tagged as complex, it means that there is a one-to-many relationship between the process and the Web services involved. This means that in order to build the workflows we will have to split this service in its constituent components. In this case, a list with more than one element will be constructed by the Ontology Reader.

Each file is generated in two phases. First, a generic template is created for each kind of file by three specific initializers: the BPEL Initializer, the WSDL Initializer and the Artifacts Initializer. After the completion of this preliminary phase, an Extension Manager is called, which will fill the file's missing fields with the appropriate information provided by the Service Builder. By analogy with the previous phase, a specific Extension Manager has been defined for each type of file. The generation of the .wsdl and the artifacts files, needed by the functional workflow, is straightforward, as information is simply copied from the java objects which were created during the Service Generation phase into the corresponding files. The .bpel files however are more difficult to generate, because the output of one service might represent the input for another one. This leads to some very intricate patterns, making it more complicated to initialize the variables before the call of a service. To solve this issue, a shift of point of view is made. First, each assign section is separated from its corresponding invoke section and they are both modeled as individual objects. As a consequence, the whole sequence section can be represented as two lists: one for the assign objects and one for the invoke objects. Secondly, each service is conscious about the assign sections that it is linked to. This way, the same service can play two roles depending on the circumstances: on one hand it can act as an output producer, while on the other, it can act as an input consumer.

Each invoke section has two associated assign sections: a pre- and a post- assign. However, this is not enough if we want to model even more complex situations. For example, the same service may act as an output producer for more than one service. To solve this issue, we need more granularity when dealing with assign blocks. This is why we have created the CopyBlock class. Objects of this type act as building blocks for an assign object or, in other words, an assign block is made up of multiple CopyBlocks. This way, the pre-assign block of any invoke section may be initialized even if each parameter of the invoked method comes from a different source. Hence, when the service acts as an output producer it fills information in the pre-assign section of the service that he is linked to. When it acts as an input consumer, it simply fills the assign section that precedes its corresponding invoke section. In order to generate the whole sequence section, we have to iterate through the list of services and allow each service to alternate its two complementary roles.

V. Fault Tolerance Support for the Workflow Engine

Our Fault Tolerance component has to be capable of addressing all three stages needed for a comprehensive management of faults. These stages refer to distinctive levels at which different actions have to be taken in order to extract as much information as possible about the errors and also provide a solution. The three stages that the Fault Tolerance component deals with are: detection, notification and recovery. When a client invokes a BPEL process, the engine creates a new instance of it and the BPEL process can be regarded as a Web Service. The communication between the BPEL process and the Web Services it invokes during its execution is done through the use of SOAP messages. SOAP messages can include the name of the Web Service to be invoked, the targeted operations and parameters. Apache Axis is the SOAP engine embedded in ActiveBPEL. It manages all the incoming and outgoing SOAP messages. Therefore, any communication between the local machine and the invoked services is intercepted by the the Axis engine.

A. Fault Detection

The strategy for detecting faults in this architecture was to intercept SOAP messages exchanged between the current executing BPEL process and the Web Services being invoked at runtime. This would ensure that if any kind of fault occurred while invoking or waiting for the result from a Web Service,

the Axis engine would catch it. Another important observation is that when an error occurs while invoking a Web Service, the service automatically builds a SOAP message which contains every available information about the error and sends the SOAP message directly to the entity that executed the invocation in the first place. In our case, the entity invoking the Web Services is the BPEL process. The ActiveBPEL engine offers a configuration file (ae-server-config.wsdd) in which we have specified a new handler for both incoming and outgoing SOAP messages from Axis. A SOAP message signaling a fault during execution has a particular structure with well defined tags. The error log handler that we have developed uses this information to detect SOAP messages indicating that a fault has occurred. The handler takes advantage of the multi-threading ability of the Axis engine being able to process concurrent SOAP message at any given time.

The primary goal of the handler is to create a log file on the local machine executing the BPEL process with any SOAP messages that indicated a fault. But managing data in this format, as well as extracting valuable information for future use is pretty difficult. Therefore, we decided to integrate a database in the architecture of the Fault Tolerance component in which all information about errors occurred during execution would be saved. Consequently, the simple error log handler has become a multipurpose mechanism which ensures real time detection of any faults and a flexible way of storing information which may prove valuable when evaluating performance and availability of Web Services. In this way, the management system can improve its heuristic of allocating and ranking resources.

B. Fault Notification

When talking about scientific applications, we should always keep in mind the fact that these types of programs can run for days or even weeks. Therefore, an important task that the Fault Tolerance component should be providing is a notification mechanism. After a fault is detected and information is saved in the database, the component should inform the user as soon as possible about the error that occurred. The idea behind the notification mechanism is to send out an email message, an RSS feed or an instant message to the client informing about any erroneous behavior of the application. Before starting the execution of a BPEL process the user can fill in a configuration file with his contact details and the error patterns he is interested in in order to be alerted by the system. The message that the user receives contains useful information about the error that occurred so that he/she can easily identify the source of the problem.

The user has the possibility of specifying the execution of certain scripts corresponding to different types of faults through the configuration file. This functionality acts as user-defined exception handling and it is not supported by many engines of its kind. As a general rule, the errors that occur at runtime can be categorized in several broad classes. Taking this into consideration, the user has the ability of specifying a particular action in case a certain error occurs.

For example, let us consider an application that is trying to determine the inverse matrix. One of the Web Services involved in computing the inverse matrix will have the task of calculating the determinant of the initial matrix. If the value of the determinant is 0, then the inverse matrix does not exist. In this case, the application will throw a fault specifying a ”Division by Zero” error. The user-defined exception handling mechanism can now intervene and the scientist might have anticipated that some of the matrices could not be inverted. Therefore, with the help of the script he/she can define another input parameter for the workflow and restart the workflow without loosing precious time. There are also many other scenarios in which this mechanism can prove extremely efficient depending on the particular functionality of the workflow. Though this functionality is pretty flexible and efficient it has one major drawback. The user, in this case the scientist has to write his own script that will be launched into execution by the Fault Tolerance component. This is not a trivial task and may prove extremely difficult in some situations. But, it also has the advantage that it can be used with any operating system and can specify almost any type of action.

C. Fault Recovery

So far, the Fault Tolerance component has the ability of detecting and notifying the user when an error occurs while executing a BPEL process. Though this is a major step in providing a fault tolerant behavior, it is not enough for a scientific application. The user can now determine the cause of the fault, has real time information provided by the notification mechanism but the only solution is to restart the workflow (Figure 3). For a process taking up four or five days this is an unacceptable solution. Therefore, the Fault Tolerance component has to provide an easier and efficient method of recovering from faults. There is one condition that the system has to meet no matter how the recovery is implemented: the partial data that might be correct has to be saved and be accessible to the user in order to make any appropriate changes. In other words, the system has to provide a checkpointing mechanism so that the computations done so far would not be lost in case of an error. Also, the system has to be capable of resuming the execution of the workflow just before the faulted service invocation.

After a thorough analysis of the ActiveBPEL engine architecture, the conclusion was that some changes have to be made in the way the engine treats faults and the mechanisms that deal with such situations. BPEL has an in-build feature which addresses the recovery stage of a faulted process called compensation. Unfortunately, the implemented design of compensation in languages such as BPEL can only conveniently be used to handle a subset of errors. The specific implementation of compensation that BPEL uses is essentially an extension of the usual exception-handling mechanisms seen in languages such as C++ or Java . But using such a mechanism requires the user to have a strong knowledge about writing BPEL processes and would definitely discourage a lot of potential users from adopting this technology. Furthermore, the current

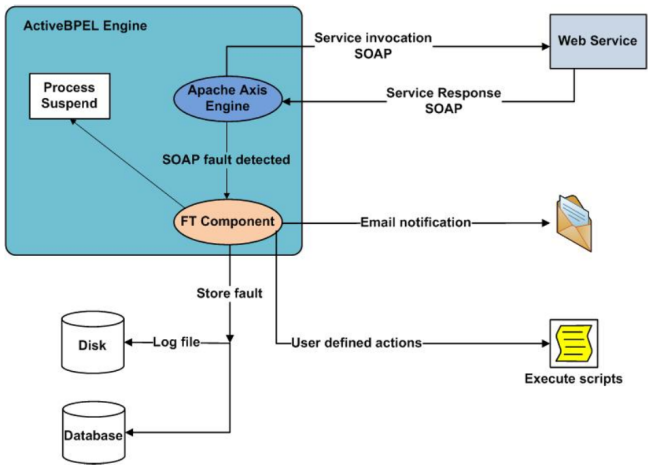

Fig. 3. The architecture of the fault tolerant component along with its triggered actions

architecture of ActiveBPEL leads to the situation in which any BPEL process has one single point of failure. By this, we understand that a workflow executing different activities in parallel will fail if just one of its activities on any of its branches fails. First of all, this is unacceptable because all the intermediary results which could be valid cannot be reused for a later submission. Secondly, occurring errors might have minor causes which could be easily dealt with, such as a Web Service being down or a bottleneck on the network.

Therefore, our approach concerning this problem was to intervene in the way ActiveBPEL deals with faults. The engine has the capability of changing the state of a process from an execution state to a suspended state and we exploited this functionality in order to deal with faults. The default behavior of the engine when it receives a fault message is to terminate the process. We have modified this behavior so that when the engine receives a fault it will suspend the process with the possibility of reactivating it. Hence, when a fault is detected in the system, the Fault Tolerance recovery mechanism has the task of suspending the corresponding process and provide the user with the ability to intervene. The suspended state is similar to a checkpoint state in which all information about the process is available. It is very important to understand that this is a local checkpoint, so no information about remote executing tasks is saved. In a SOA architecture the majority of actions will involve service invocation and as a consequence, the highest probability for a fault to appear will be during the invocation sequence. The fact that the Fault Tolerance component provides access to the parameters and the endpoints of the Web Services invoked gives the user a better control over the entire process. If an error occurs during the invocation of an Web Service then the corresponding process will be immediately suspended. The user will have access to all the information that is directly linked to the last invocation which failed.

A secondary goal to building and integrating a Fault Tolerance component in an open-source BPEL engine was to implement this in a flexible and maintainable way. That is why we decided to use the Aspect Oriented Programming (AOP) [8], [10] paradigm to explore a different approach to the problem of software extension and concentrated on introducing the new functionality as aspects of the base system. The Fault Tolerance component is made up of two distinctive modules: the module responsible for detecting the faults and providing the notifications and the module designed to recover the workflows without loosing the computations done so far. The first module is implemented as an extension of the Apache Axis engine in the traditional manner while the second module is implemented using Aspect-Oriented Programming. AOP fosters the goal of separation of concerns. The AOP technology emerged for modularizing crosscutting concerns. Classical examples of crosscutting concerns are: logging, security or exception handling [9]. Crosscutting concerns are concerns (aspects) that can not be encapsulated into single components. On the contrary, the implementation of these concerns crosscut the software structure of a system. Therefore, this paradigm is ideal for implementing our Fault Tolerance component because it is much easier to develop and understand the necessary modification that need to be integrated in the workflow engine.

VI. The Scheduling Component

As mentioned before, the DyAG scheduler is a subsystem of the project, which manages the mapping between the requests from the upper layer and the concrete Web Services available on Grid. The resources are described using an information model which is not bound to a particular implementation, offering enhanced flexibility. The decentralized architecture is presented in Figure 4.

As it can be observed, the DyAG scheduler interacts directly with the EGEE middleware by implementing a LDAP interface and connecting to one or more BDII sites. DyAG architecture has two main subsystems, encapsulated in the DyAG object: the Grid Monitor, responsible for extracting information and modifying configuration of the Grid engine and the Schedule Monitor, responsible for fair mapping of the userss requests and the services available. The DyAG announces its methods and connects with its clients through a RMI interface. Therefore, the DyAG client can invoke the methods exposed in the RMI interface in order to satisfy users requests.

Typically, the user installs the executable and starts issuing commands. As it was mentioned before, the client can submit jobs and choose from the available policies the scheduler can implement the most suitable one for his task. The DyAG has a default configuration and therefore a default policy for solving the jobs that were submitted, but this configuration can be modified if the user specifies it. Other methods included in the RMI interface are for administrative purposes and allow clients to start / stop monitoring, discover the services available and report the eventual faults to the DyAG [7].

These development comes naturally from the design of the DyAG module is the separation of the Scheduling and Monitoring sub modules, which could become themselves web services. This could greatly increase the scalability of the

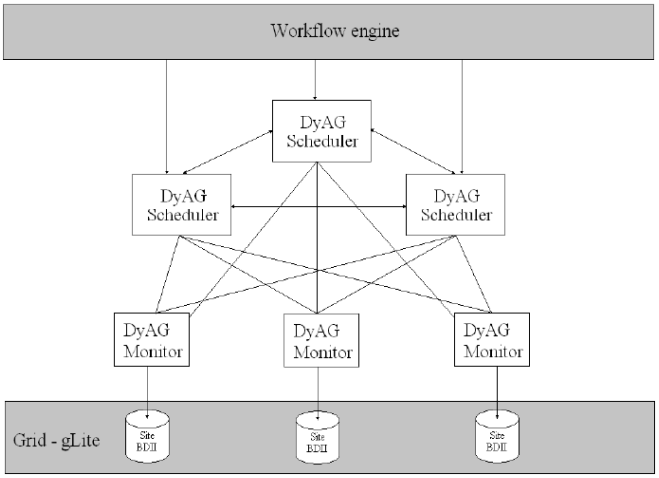

Fig. 4. A distributed DyAG scheduling architecture

solution, by having multiple schedulers which could either cooperate for the finding of a better solution, or could serve as backups in case one of them crashes, or could improve the performance of the system when serving many requests from one or more workflow engines.

In this scenario, there could be a monitoring module deployed at each site where there are web services waiting for jobs, thus reducing the communication latency with the respective BDII servers. Because of this and through the employment of whitelists in order to monitor only the relevant web services, the performance of the system would increase. Whereas now there is only one Monitoring module which performs large queries on off-site BDIIs in order to retrieve information about all the relevant resources, in the new scenario, the schedulers would query the off-site Monitoring services, and retrieve the information about specific services, those that have been requested by the upper layer. The figure below describes this distributed scheduling architecture.

VII. Illustration

The application used for illustration is represented by parallel matrix-matrix multiplication. Effective design of parallel matrix multiplication algorithms relies on the consideration of many interdependent issues based on the underlying parallel machine or network upon which such algorithms will be implemented, as well as, the type of methodology utilized by an algorithm. In [9] it was determinate the parallel complexity of multiplying two (not necessarily square) matrices on parallel distributed-memory machines and/or networks. Assume that communication follows the linear model $t(k) = t_{lat} + kt_{bw}$, where k is the number of double words sent, t_{lat} is the latency time, and t_{bw} is the time associated with the bandwidth of the communication channel. Each computation step will require $(2n/p) * (n/p) * n = (2n^3)/p^2$ flops. Each task in workflow involves $t = t_{lat} + (n^2)/p * t_{bw}$ communication time. Notice that if the total time consisted only of the first term, then the algorithm has perfect speedup - since it is the total number of flops (which we are assuming take one time unit each) divided by the number of processors. Hence the other two terms give the overhead due to parallelism. The theoretical speedup is then obtained by dividing the above time into $2n^3$, and the parallel efficiency is obtained by dividing speedup by p for [13]:

$$E = \frac{1}{1 + \frac{p(p-1)t_{lat}}{2n^3} + \frac{(p-1)t_{bw}}{2n}}.$$

The cost associated with the previous computation is $O(n^3)$, while the amount of communication is $O(n^2)$. This is the key to another definition of a "scalable" distributed memory algorithm; the communication costs grow slower than the computation costs. Here, however, the definition of scalable has undergone a subtle shift from that given earlier where it meant "parallel efficiency is bounded below by a positive number, as p grows". This second version is probably the one most researchers use since it is a weaker requirement and more easily verified.

In conclusion, the iterations in the process of parallel matrix-matrix multiplication represents a good illustration for scientific applications execution in Grid environment.

VIII. Performance Evaluation

The main advantages which would come with the implementation of the previously described distributed architecture are the increased potential for scalability, greater fault tolerance through replication, and the elimination of the single point of failure and performance bottleneck which characterize a non-distributed architecture.

To demonstrate the effectiveness of the algorithm, we considered the following scenario: we start 5 DyAGs and 51 clients with these settings: first 2 clients use SET, 21 clients use `NO_SET_RANDOM` and 28 clients use `NO_SET_UNIQUE`. As stated before, the OLB algorithm takes into account the current number of tasks a DyAG has to complete and the number of clients mapped to that DyAG. Suppose we first start the clients with option SET. They will be mapped to the DyAG4 and DyAG5 respectively. Next, the clients with the option `NO_SET_UNIQUE` will be started. They will be mapped to the DyAG4, DyAG3 and DyAG2 respectively, because the rest of the DyAGs have their default configuration altered. After that, we start the clients with the option `NO_SET_RANDOM`. They will be mapped to all DyAGs, because they do not take into account the scheduling policy currently in use (see Figure 5).

We consider the same scenario presented before. First start the clients with the SET option and map them to the DyAG4 and DyAG3 respectively. Start 18 clients with the `NO_SET_UNIQUE` option and map them to DyAG3, DyAG2 and DyAG1 respectively. Suppose DyAG1 fails. The six clients this DyAG was assigned to will be re-mapped to DyAG2 and DyAG3. Next, start the last 10 clients with the option `NO_SET_UNIQUE`. They will be mapped to DyAG2 and DyAG3 too. Now start the clients with the option `NO_SET_RANDOM`. They will be mapped to the DyAG4 and DyAG5. Like in the previous case, it can be observed

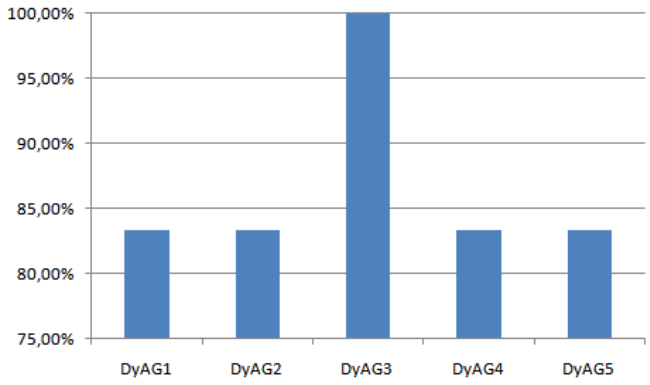

Fig. 5. Load Balancing without Faults

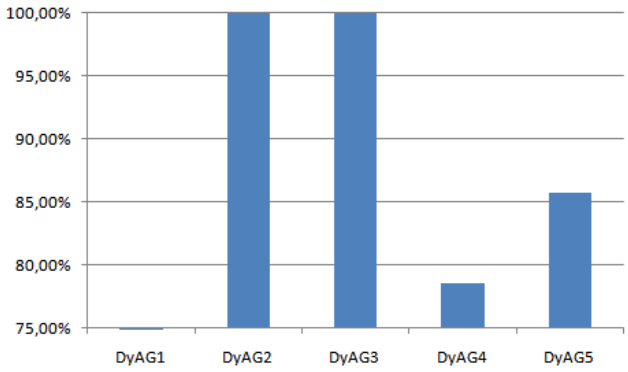

Fig. 6. Load Balancing with Faults (DyAG Fails)

that clients are assigned to DyAGs in a load-balanced way, therefore ensuring maximal utilization of resources (see Figure 6).

IX. Conclusion

Resource management in large scale distributed systems faces some major challenges among which are: the heterogeneity and the autonomy of the local sites, the high dynamism of distributed systems and the separation of computational resources from data storage resources. In addition to these, workflow management systems also have to cope with complex applications, that contain large numbers of inter-dependent tasks. Taking these aspects into consideration, in order to improve resource and workflow management platforms it is essential to understand how they perform in "real world" conditions, in a large scale distributed system.

In this paper we addressed these issues by building a workflow platform targeted at scientific applications. Our solution provides an interface for abstract workflow specification which translates an workflow description in the application's semantics to BPEL, handles failures transparently and is able to make efficient schedules of tasks to the available resources. Our tests proved better reaction times when executing the workflows with our enhanced solutions. A future key interest will be developing a benchmark based method for evaluating the performances of our workflow platform to similar ones.

References

[1] Khalid Belhajjame, Katy Wolstencroft, Oscar Corcho, Tom Oinn, Franck Tanoh, Alan William, and Carole Goble. Metadata management in the taverna workflow system. In *CCGRID '08: Proceedings of the 2008 Eighth IEEE International Symposium on Cluster Computing and the Grid*, pages 651–656, Washington, DC, USA, 2008. IEEE Computer Society.

[2] Justin Cappos, Scott Baker, Jeremy Plichta, Duy Nyugen, Jason Hardies, Matt Borgard, Jeffry Johnston, and John H. Hartman. Stork: package management for distributed vm environments. In *LISA'07: Proceedings of the 21st conference on Large Installation System Administration Conference*, pages 1–16, Berkeley, CA, USA, 2007. USENIX Association.

[3] Ewa Deelman, Gurmeet Singh, Mei-Hui Su, James Blythe, Yolanda Gil, Carl Kesselman, Gaurang Mehta, Karan Vahi, G. Bruce Berriman, John Good, Anastasia Laity, Joseph C. Jacob, and Daniel S. Katz. Pegasus: A framework for mapping complex scientific workflows onto distributed systems. *Sci. Program.*, 13(3):219–237, 2005.

[4] James Frey, Todd Tannenbaum, Miron Livny, Ian Foster, and Steven Tuecke. Condor-g: A computation management agent for multi-institutional grids. *Cluster Computing*, 5(3):237–246, 2002.

[5] Bernardo Cuenca Grau, Ian Horrocks, Boris Motik, Bijan Parsia, Peter Patel-Schneider, and Ulrike Sattler. Owl 2: The next step for owl. *Web Semant.*, 6(4):309–322, 2008.

[6] Andrew Harrison, Ian Taylor, Ian Wang, and Matthew Shields. Ws-rf workflow in triana. *Int. J. High Perform. Comput. Appl.*, 22(3):268–283, 2008.

[7] Marius Ion, Florin Pop, Ciprian Dobre, and Valentin Cristea. Dynamic resources allocation in grid enviroments. In *SYNASC '09: Proceedings of the 2009 11th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, pages 213–220, Washington, DC, USA, 2009. IEEE Computer Society.

[8] Gregor Kiczales and Mira Mezini. Aspect-oriented programming and modular reasoning. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 49–58, New York, NY, USA, 2005. ACM.

[9] Eunice E. Santos. Parallel complexity of matrix multiplication. *J. Supercomput.*, 25(2):155–175, 2003.

[10] Friedrich Steimann. The paradoxical success of aspect-oriented programming. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 481–497, New York, NY, USA, 2006. ACM.

[11] C. Stratan, A. Iosup, and D. H. J. Epema. A performance study of grid workflow engines. In *GRID '08: Proceedings of the 2008 9th IEEE/ACM International Conference on Grid Computing*, pages 25–32, Washington, DC, USA, 2008. IEEE Computer Society.

[12] Wei Tan, Paolo Missier, Ian Foster, Ravi Madduri, David De Roure, and Carole Goble. A comparison of using taverna and bpel in building scientific workflows: the case of cagrid. *Concurr. Comput. : Pract. Exper.*, 22(9):1098–1117, 2010.

[13] Ahmed S. Zekri and Stanislav G. Sedukhin. Computationally efficient parallel matrix to matrix multiplication on the torus. In *ISHPC'05/ALPS'06: Proceedings of the 6th international symposium on high-performance computing and 1st international conference on Advanced low power systems*, pages 219–226, Berlin, Heidelberg, 2008. Springer-Verlag.