Design and Implementation of a Fault Tolerant Job Flow Manager Using Job Flow Patterns and Recovery Policies

Selim Kalayci[1], Onyeka Ezenwoye[2], Balaji Viswanathan[3], Gargi Dasgupta[3], S. Masoud Sadjadi[1], and Liana Fong[4]

[1] Florida International University, Miami, FL, USA
{skala001, sadjadi}@cs.fiu.edu
[2] South Dakota State University, Brookings, SD, USA
onyeka.ezenwoye@sdstate.edu
[3] IBM India Research Lab, New Delhi, India
{gdasgupt, bviswana}@in.ibm.com
[4] IBM Watson Research Center, Hawthorne, NY, USA
llfong@us.ibm.com

Abstract. Currently, many grid applications are developed as job flows that are composed of multiple jobs. The execution of job flows requires the support of a job flow manager and a job scheduler. Due to the long running nature of job flows, the support for fault tolerance and recovery policies is especially important. This support is inherently complicated due to the sequencing and dependency of jobs within a flow, and the required coordination between workflow engines and job schedulers. In this paper, we describe the design and implementation of a job flow manager that supports fault tolerance. First, we identify and label job flow patterns within a job flow during deployment time. Next, at runtime, we introduce a proxy that intercepts and resolves faults using job flow patterns and their corresponding fault-recovery policies. Our design has the advantages of separation of the job flow and fault handling logic, requiring no manipulation at the modeling time, and providing flexibility with respect to fault resolution at runtime. We validate our design with a prototypical implementation based on the ActiveBPEL workflow engine and GridWay Metascheduler, and Montage application as the case study.

1 Introduction

Currently, many complex computing applications are being developed as job flows that are composed of multiple lower-function jobs. The execution of these job flows requires functional support of a job flow manager and a job scheduler. Due to the typical long running nature of job flows, the support for fault tolerance and recovery policies is especially important, and requires coordination between workflow engines and job schedulers. Very often, the failure of a job within a flow cannot be treated in isolation and recovery actions may need to be applied to preceding and dependent jobs as well. The interaction of multi-layered grid services with dynamic distributed resources makes fault-tolerance a critical and challenging aspect of job flow management. In this paper, we address fault tolerant issues at runtime using recurrent job flow patterns, fault tolerant patterns, and a transparent proxy.

A. Bouguettaya, I. Krueger, and T. Margaria (Eds.): ICSOC 2008, LNCS 5364, pp. 54–69, 2008.

Many exemplary job flows can be found in grid and cluster computing environments, such as the Montage application [1], workflows[1] in e-Science [2], and many commercial job flows using Z/OS JCL [3]. Execution of job flows requires functional support of a job flow manager and a job scheduler, as either two components of an integrated software or two separate software components. One approach to handle flow-level compensation is to include failure management logic during the development of job flow model. This approach adds the complexity of fault recovery logic to the application flow. Wei et al. [4] investigated how to incorporate fault handling and recovery strategy for jobs at modeling time without requiring the user to embed the recovery logic in the application flows by using a two-staged methodology. However, their work requires modification of the original flow to incorporate fault-handling policies at modeling time. An alternate approach is to handle workflow failures at runtime, without explicit changes to workflow process logic. In TRAP/BPEL [5], an intermediate proxy traps calls from the workflow engine, and on behalf of it, implements a runtime failure handling approach for stateless web services. However, unlike stateless web services, defining recovery policies for jobs is more challenging, as different jobs may fail at different stages of execution and may require different types of recovery actions. Long-running jobs often may require elaborate cleanup phases on account of failure.

In this paper, we present the architecture of a fault-tolerant job flow manager using job flow patterns and web services. A salient feature of our architecture is that the fault-handling for job flow execution can be introduced as late as , requiring no modifications to workflows during the modeling time. Thus, our design also has the advantages of the separation of job flow and fault handling logic, and flexibility in fault resolution at runtime. Our technique first examines the workflow automatically during deployment time to identify and label common, recurrent job flow patterns based on a knowledge base that incorporates up-to-date job flow patterns [6]. Next, we introduce a proxy that *transparently* intercepts and monitors job submissions and resolves faults at runtime using user-defined recovery policies, the identified job flow patterns, and their corresponding fault-tolerant patterns. Depending on the recovery policies, recovery actions will be selected from the alternative fault-tolerant patterns during runtime. To validate our design, we implemented a job flow management system using the open-source software, including ActiveBPEL [7] workflow engine and GridWay Meta-scheduler [8]. We used the Montage application as the case study.

The rest of this paper is organized as follows. Section 2 provides a brief background on job flow and fault tolerant patterns. Section 3 provides an architectural overview of our job flow management system. Section 4 further elaborates on the job flow management design. Section 5 introduces our prototypical implementation. Section 6 presents our experimental results. Section 7 surveys related work. Section 8 concludes the paper with a short summary of our work and suggests some directions for future work.

2 Job Flow and Fault Tolerant Patterns

Workflow failures have been broadly categorized as work item failures, deadline expiry, resource unavailability, external triggers, and constraint violation [9]. Some of

[1] In this paper, the terms job flows and workflows will be used interchangeably.

these failures can be handled well by specifying recovery actions at modeling time. However, in an uncontrolled grid environment, exceptions may occur due to a variety of reasons. Handling all this at modeling time is infeasible due to the high complexity it will add to the workflow and the pre-knowledge of all different failure scenarios that can arise. Prior literature [10] characterizes workflow exceptions for web services into a set of patterns. These patterns identify the individual task that the exception is based and all other dependent tasks that need to be handled; and specify the recovery action to be undertaken. In [6], we classified grid failures into some common patterns and identified common grid fault-tolerance patterns to be applied to them.

2.1 Job Flow Patterns

Below, we briefly summarize the relevant abstract, reusable patterns presented in [6], which arise in a job flow management system. The patterns are related to the submission of jobs from the job flow manager to the job scheduler, the exchange of monitored information regarding job states between the entities, the staging of data required for these jobs, and their execution on the scheduler resources.

Job Submission and Monitoring. A job submission by the flow manager to the job scheduler involves invoking the corresponding job scheduler interfaces to perform the functions of submission of the job to the resource management layer and monitoring for any state changes. Examples of different submission patterns are synchronous job submission and asynchronous submission with polling/notification.

Data Staging. Many grid jobs require input data, and in the absence of a shared file system, these datasets need to be staged in at the site of execution. A typical data staging pattern in job flows comprises staging in data from either producer jobs or from defined inputs, followed by a job submission pattern.

Job Execution. Job execution completion status is captured in the job state and in the job state transitions. Some job execution failures are best handled by resubmitting the job either at the same domain or redirecting the job to a new domain. Others may require additional handling such that getting information from the job definitions.

2.2 Fault Tolerant Patterns

Fig. 1 shows a state transition diagram that models the patterns identified in Section 2.1. A failure in any one or more of these activities entails a transition to the *Failed* state and each such transition represents a fault-pattern. Thus, the specific fault-patterns observed due to failure at the resource management layer can be classified broadly as job submission failure, data staging failure, job execution failure, job notification failure, and job query failure. For each of these faults, we have outlined recovery actions that are generically repeatable and can be applied as fault tolerant patterns [6]. Some of the identified *fault tolerant patterns* include:

Retry job. A job is re-submitted for execution upon the occurrence of an exception during job submission or execution. In this pattern, jobs are submitted to the same domain. The resubmission of the job may require modifications in the job specification and resource requirements. For example, submission failures that arise from the

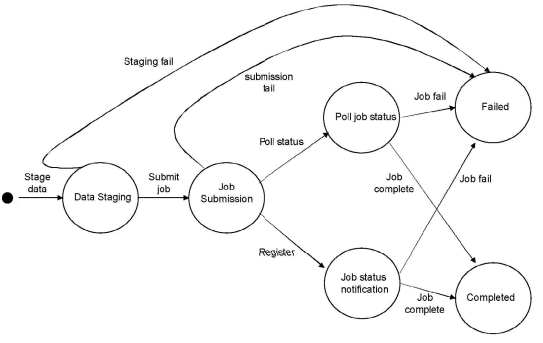

Fig. 1. Statechart diagram capturing the typical job flow patterns

temporary unavailability of the scheduling service can be recovered by re-submitting the job whereas execution errors require more detailed analysis of job state, status, and exit codes as well as a significant amount of domain expertise for fault-handling. When a job is re-submitted, the job flow manager automatically needs to re-poll or re-register for the new submission.

Redirect job. A job is redirected to a different domain for execution upon the occurrence of an exception during job submission or execution. This fault-tolerant pattern may be selected because all of its previous attempts of re-submitting to the same domain have failed or because it has been decided that the submission of the job to the current domain has a low probability of success. The input data of the redirected job is re-staged at the new target domain. Output data may need to be staged out.

Retry query. Polling for job status and/or registration for job status notification is resumed upon job re-submission. The newly submitted job is transparently re-polled (possibly using the newly returned job ID). This involves translating and modifying the original polling messages from the flow manager to map to the re-polling of the newly re-submitted job.

Force-fail. When no further progress is possible, the job state is changed to Failed**.**

3 Architecture Overview

In this section, we introduce the architecture of our fault-tolerant job flow manager, which is depicted in Fig. 2. This architecture incorporates two distinct advantages with respect to other related works. The first advantage is to avoid introducing the complexity of the recovery process to the application flows. Hence, the approach would maintain the separation of concerns. Please note that in Fig. 2, the components and arrows with solid line indicate those portions of the architecture related to the core business logic of the job flow and those with dashed lines relate to the fault-tolerance and recovery portion of the job flow. Secondly, the approach provides the flexibility of defining job flow patterns, fault-tolerant patterns, and recovery policies, which may be stored unto the knowledge base of the system, as late as deployment

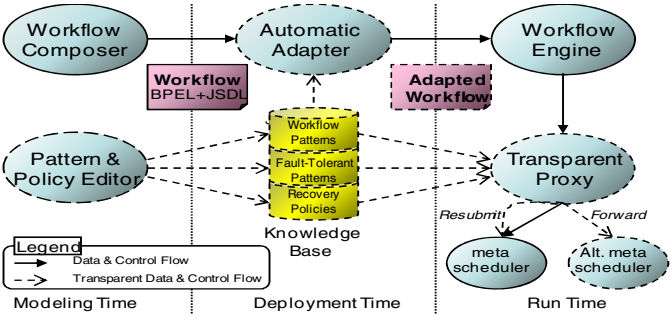

Fig. 2. The architecture of our fault-tolerant job flow manager

and runtime. The sophistication of recovery policies can grow with the knowledge of the flow and fault patterns. In addition to the above advantages, our architecture follows the two-layer design for job flow orchestration and scheduling introduced in [6]. The job flow manager is responsible for execution of the job flow according to the specified logic while being agnostic to resource allocation decisions. In contrast, the job scheduler component matches user work requests to grid resources, performs allocation, and enables distributed execution of the individual jobs in a workflow.

We note that there are many custom job flow and job definition languages used in the literature to express job flow and individual jobs. The OASIS's Web Service Business Process Execution Language (WS-BPEL or BPEL) [11] has attracted many researchers to explore for job flow specification, as there are numerous implementations (*e.g.* ActiveBPEL [7] and Websphere Process Server [12]). Thus, BPEL is used to express the flow of jobs, their dependencies, and flow of data among the jobs. A job definition describes a unit of job (*e.g.* an executable file together with parameters and resource requirements) to be submitted to a scheduler. The GGF's Job Submission Description Language (JSDL) [13] has been accepted by many researchers as the de-facto job language. Without loss of generality, our design uses BPEL and JSDL as our reference languages for job flow and job definition.

As illustrated in the left side of Fig. 2, first, a domain expert will use the *Workflow Composer* to specify the business logic of the application using BPEL+JSDL. The domain expert should only be concerned about the business logic of the application and should not be concerned about handling faults and exceptions. We note that a BPEL+JSDL workflow is still a valid BPEL. The job descriptions are treated as complex types in XML, which in turn are used as the parameters to some *Invoke* constructs in BPEL. Therefore, any BPEL editor can be used as the workflow composer. We also note that such editors may not have the capability to compose job descriptions in JSDL, but it is possible to compose job descriptions in JSDL in other editors and then copy/paste the corresponding XML document.

During deployment time, the resulting workflow is passed to the *Automatic Adapter,* which in turn automatically generates a functionally equivalent workflow with the context information that is needed for the Proxy to monitor the interaction between the flow manager and the meta-schedulers. The automatic adapter has an algorithm that identifies the known workflow patterns (*e.g.* job submission) within the workflow. The most updated workflow patterns are stored in the *Knowledge Base*.

New workflow patterns can be added to the knowledge base using the *Pattern & Policy Editor*. The generated workflow, called *adapted workflow,* would then include the context information (*e.g.* the *pattern* and *job id* - details to be provided in the next section), but it does not have any knowledge of how to handle faults at runtime. Instead, the adaptation incorporates some *generic* interceptors at sensitive join-points in the original BPEL+JSDL workflow. The most appropriate place to insert interception hooks in a BPEL+JSDL workflow is at the interaction join-points (i.e. at scheduler service invocation). The inserted code is in the form of standard BPEL constructs to ensure the portability of the modified process. This adaptation permits the BPEL+JSDL workflow behavior to be modified at runtime by the *Transparent Proxy*.

At runtime, the BPEL+JSDL workflow will be executed by the *Workflow Engine.* The workflow engine can be any BPEL engine without any modification or extension, as we did not extend BPEL in our work. Note that during the automatic adaptation of the workflow, all the calls originally targeted for the local *Meta-scheduler* are redirected to the *Transparent Proxy [6]*. Therefore, the Transparent Proxy will intercept all the calls to the Meta-scheduler. The Proxy will appear as a Meta-scheduler to the workflow process, and as a workflow process to the Meta-scheduler; hence, the name *transparent*. Its main responsibility includes submission of the intercepted jobs to the local Meta-scheduler and notifying the workflow process of the job status when it receives job status updates from the Meta-scheduler. In addition, it implements a pattern-matching algorithm that monitors the behavior of the intercepted calls and provides fault-tolerant behavior when faults occur. The algorithm is based on the classification of exception handling introduced in Section 2, the *Recovery Policies*, the context information embedded in the adapted workflow, the *Workflow Patterns*, and their corresponding *Fault-Tolerant Patterns*. For example, following the recovery policies governing the current faulty situation, the Transparent Proxy may resubmit the job to the same Meta-scheduler or redirect it to another Meta-scheduler.

4 Detailed Design

In this section we provide the detailed design of our fault-tolerant job flow manager.

Job flow Adaptation. Fig. 3 shows a code snippet of an example job flow that includes a job submission invocation to the underlying Meta-scheduler. Prior to the invocation construct, the JSDL definition for the job is copied to the input variable (*jobReqMsg*) of the Meta-scheduler. This invocation is replaced during the adaptation process by the Automatic Adapter with a corresponding invocation to the Transparent Proxy; thus, the invocations meant for the Meta-scheduler will be intercepted by the Proxy. Note that the input message for Meta-scheduler is now sent to the Proxy. Fig. 4 shows the same section of the code as in Fig. 3 after the adaptation process. In Fig. 4, the invocation to the Meta-scheduler is replaced with that of the Proxy. Note that invocations to the Proxy have an additional parameter, which is the job identifier (*jobID*). The Proxy uses this jobID to monitor and maintain the state of each job from inception to completion; in other words, from data-staging to completion as depicted in Fig. 1. This unique jobID helps the Proxy to keep track of the job as it progresses through each pattern.

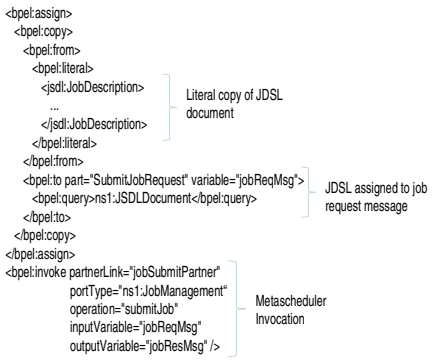

Fig. 3. Job flow snippet code prior to adaptation

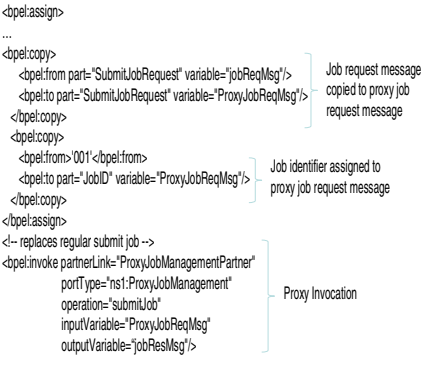

Fig. 4. Job flow example snippet code after the adaptation: invocation to the Proxy

Interface Design. In [14], we introduced a set of APIs for meta-schedulers that supports interoperability among different meta-schedulers and demonstrated how jobs originating from a meta-scheduler can be scheduled to run on resources under the control of other meta-schedulers. In this work, we utilize this common meta-scheduler interface for communication between the Workflow Engine, Transparent Proxy, and Meta-scheduler. The job submission interface provided by the underlying Meta-scheduler layer, defines the following operations. The *submitJob* operation submits a job to the Meta-scheduler in JSDL format. If the job gets successfully submitted, the Meta-scheduler returns the corresponding job id for this job. The *getProperties* operation queries the status of a certain job by specifying its job id. The *queryJobs* operation queries the status of all submitted jobs during the current session. Certain filters for the jobs to be queried can be provided optionally. The *cancelJob* operation cancels the execution of a certain job specified by its jobID.

The Transparent Proxy interface extends the interface of the Meta-scheduler. As mentioned before, the operations of the Proxy interface require an additional parameter for job identification (*jobID*). The Proxy, acting as an intermediary between the Workflow Engine and the Meta-scheduler, requires the jobID in order to keep track of the interactions between the Workflow Engine and the Meta-scheduler; the intended pattern is implied by the invoked operation (*e.g. submitJob* operation for *job submission* pattern and *queryJobs* for *poll job status* pattern). The Proxy also maintains state information about individual jobs.

Fault Handling and Recovery Policies. On receipt of an invocation, the Proxy stores any necessary information and redirects the call to the local Meta-scheduler. Any reply from the Meta-scheduler is routed through the Proxy. In the event of a fault, the Proxy can enact fault-tolerance actions as defined in the recovery policy. The sequence diagram in Fig. 5 shows the interactions between the Workflow Engine, Transparent Proxy, Meta-scheduler, and Alternative Meta-scheduler during a job submission.

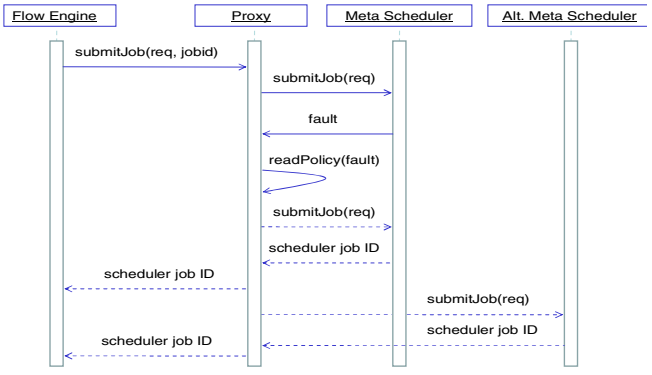

Fig. 5. The interactions between Workflow Engine, Transparent Proxy, Meta-scheduler, and Alternate Meta-scheduler during job submission

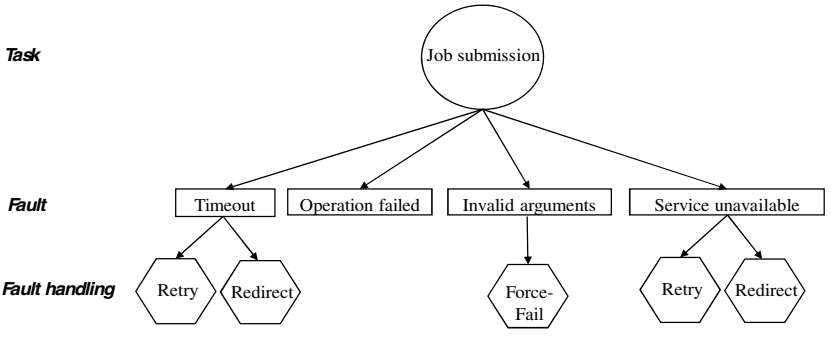

Fig. 6. A graphical representation of an example recovery policy for job submission

As shown in the figure, the Proxy redirects a job submission call to the Meta-scheduler and initiates a timer to measure the time of the invocation; the duration of the timer is defined in the recovery policy for this particular invocation. In the event of a fault (or timeout), the Proxy will either retry the job submission to the same Meta-scheduler or redirect the job to an alternative Meta-scheduler. The Proxy consults the recovery policy to determine what action to take. The number of retries, length of retry interval, and maximum number of attempts to redirect the job may be defined in the recovery policy.

Fig. 6 shows a graphical model of an example recovery policy where a set of possible faults are associated with the job submission task. Recovery actions are defined for those specified fault events. For example, a retry of job submission is specified for a timeout fault. If the retries are exhausted for the same fault, the Proxy is then required to redirect the job submission to an alternative Meta-scheduler. A failed operation follows its normal fault handling process in the workflow and the Proxy does not get involved in this situation. A Force-Fail (section 2.2) is enacted for an "*invalid arguments*" fault and a Retry/Redirect operation is required if the desired Meta-scheduler is unavailable. Similar recovery policy specifications can be specified for

the data-staging and job-status-polling stages of the job. This policy model can be easily represented in XML within the policy document. The advantages of this model are: (1) simplification of representation; (2) ease of extension; and (3) flexibility since recovery actions do not have to be hardcoded in the Proxy.

5 Prototypical Implementation

This section presents the details of our fault-tolerant job flow management prototype setup at Florida International University (FIU) that consists of a job flow manager, a Proxy and a scheduler component. For building this testbed, we used the ActiveBPEL flow engine (v4.1), and the Meta-scheduler built based on the Globus Toolkit (GT4) [15], and GridWay Meta-scheduler (v5.2.3). We used the DRMAA [16] API for job submission, monitoring, and controlling.

To detect faults during job submission, job monitoring, or job execution, the Proxy maintains an internal state machine and several timers for each job. The state machine at the Proxy is same as the one shown in Fig. 1. On fault-detection, the Proxy consults its policy knowledge base and takes the necessary actions specified in the policy file. Policies, defined in XML, are generic and thus apply to all jobs. The snippet of a generic policy file used by the Proxy is shown in Fig. 7.

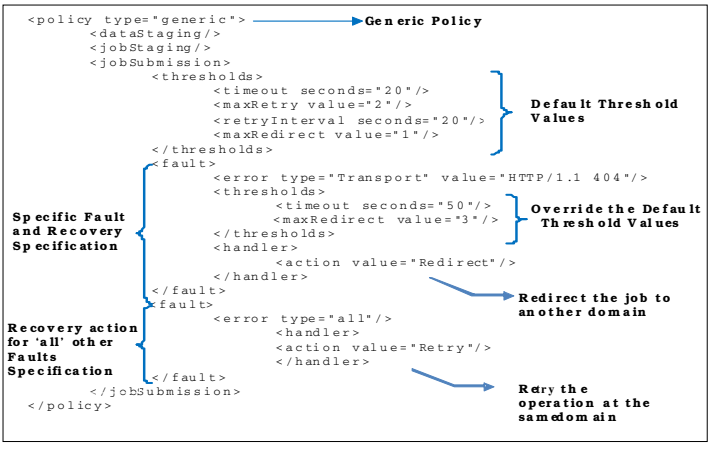

Fig. 7. A generic policy example snippet

In this policy example, a list of fault handlers can be specified (*fault* elements in Fig. 7), each capable of handling a specific fault (*error* elements). The fault handlers are matched in sequence and the first matching fault handler is applied. Fault handling is done by performing the associated sequence of recovery actions (*action* elements). A default handler which matches all faults (*error type*="*all*") can apply a generic set of recovery actions for all unmatched faults.

We used the Montage application [1] for this experimentation. The application structure, as shown in Fig. 8, consists of the computational workflow of re-projection of input images (*mProject*), modeling of background radiation (*mDiffFit*), rectification of images (*mBackground*) and co-addition of re-projected, background corrected images into a final mosaic (*mAdd*). Activities like *mProject* and *mDiffFit* can run as parallel tasks. The most computationally intensive step is that of *mProject* while the most data-intensive steps are that of *mOverlaps* and *mBackground*. The inherent parallelism among jobs in its different stages makes Montage a very suitable candidate application for grid enablement.

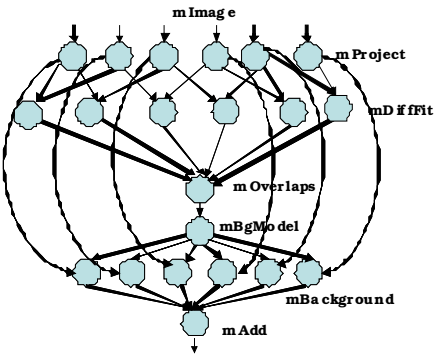

Fig. 8. Montage Application

6 Experimental Results

We setup our Montage application execution environment on two different platforms: the first, called GCB, is an eight node cluster, with dual P4@3GHz and 1 GB of memory per node, running GT4, Rocks, and CentOS 4.4. The second, called Skywarp, is a single node P4 with 1 GB of memory as the alternative execution environment. Both environments have the same Meta-scheduler setup and Montage executables and libraries.

6.1 Proxy Overhead and Opportunistic Behavior Analysis

When there is no fault to be handled by the Proxy, the Proxy intercepts all interactions between Flow Engine and the Meta-scheduler; causing a small amount of overhead during job flow execution. To calculate this performance overhead, we executed the same workflow on GCB, both with and without our fault tolerant infrastructure. Table 1 presents the average statistics from 5 separate runs without any faults during the execution. As indicated in the column with the heading "No Slowdown", the Proxy introduces a very small overhead in execution time.

In some cases the presence of a Proxy may not be necessary for the successful completion of the workflow. However, it may provide better performance (depending on the system load at the time). One such scenario is the slowdown during the service call directed towards the Meta-scheduler, resulting in a long delay before getting back

Table 1. Average Montage workflow runtime table on GCB with/without Proxy

	No Slowdown	1 Slowdown	2 Slowdowns
No Proxy	18 min. 44 sec.	19 min. 12 sec.	19 min. 43 sec.
With Proxy	18 min. 46 sec.	19 min. 01 sec.	19 min. 14 sec.

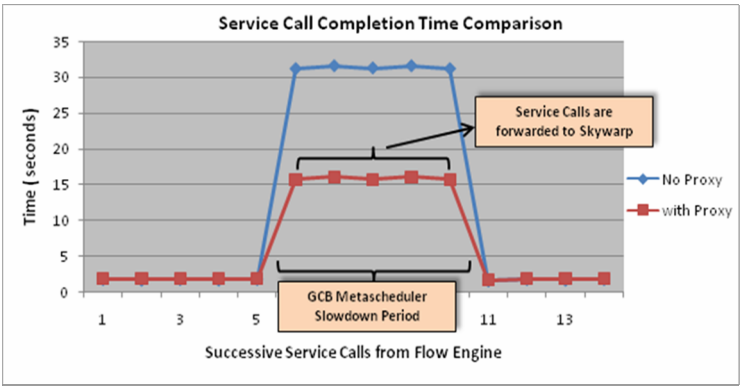

Fig. 9. Average total service calls completion time for successive calls

with a reply. In this case, the Proxy can detect the slowdown in the Meta-scheduler service and redirect the requested service to the alternative Meta-scheduler, which is idle at the time. We call this the *opportunistic behavior* of the Proxy. For the class of short-running jobs, we present results for two such slowdowns in Table 1. Each slowdown lasted for 30 seconds, while the Proxy's call timeout value was configured at 10 seconds. The recovery policy was set to Redirect. Table 1 (columns 2, 3) demonstrates the opportunistic behavior of Proxy that results in shorter runtimes in case of slowdown/interruption in service. Figure 9 shows the series of individual request completion time with and without the Proxy.

6.2 Fault-Recovery Scenarios and Experimental Results

This section details the specific fault and recovery scenarios we experimented with while inducing faults in our test-bed for the Montage workflow:

1. Job submission faults:

    a. The particular web service is not available and the service returns a transport level error (such as HTTP 503: *Service Unavailable* Message).
    b. The submit request message gets to the Job Submission Web Service, but the service internally decides to send a Service Unavailable Fault message.
    c. Timeout value that matches for the current job is exceeded.

Proxy Action: Retry the job request on the same domain for a maximum of *N* times, with *M* seconds interval between each retry. If after *N* retries job submission doesn't succeed, try another fault-tolerant pattern (*i.e.*, Redirect).

2. Job status query faults:

    a. An "UnknownResourceFault" message is returned by the operation "getProperties". This may occur due to an unknown job id (*i.e.*, a job id not recognized by Meta-scheduler).

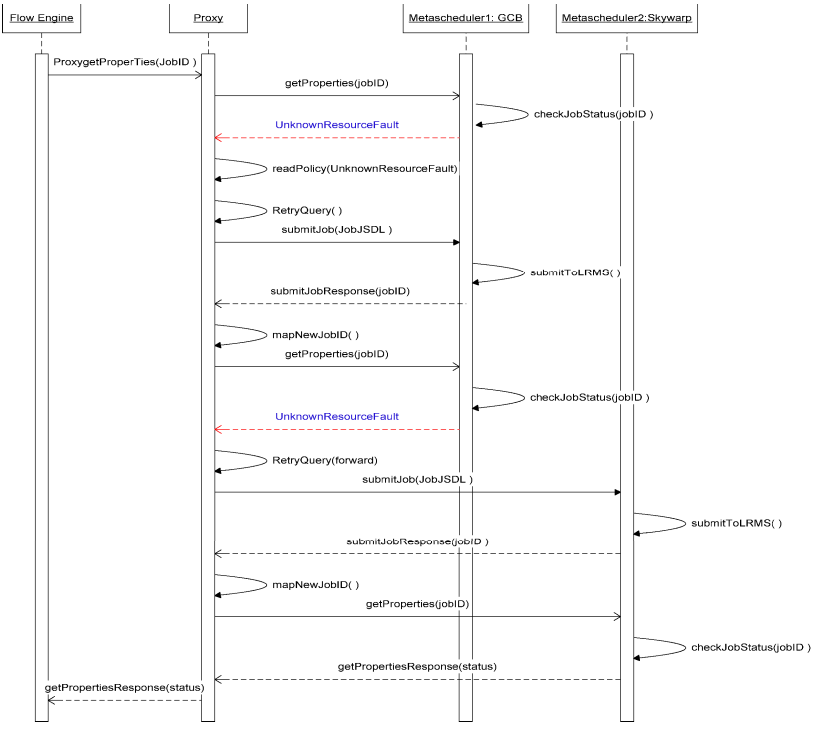

Fig. 10. Sequence diagram for scenario 2.a. Proxy first enacts the "Retry Query" on the same domain (GCB), then on another domain (Skywarp).

Proxy Action: Retry the corresponding job on the same domain. Again, the policy can specify the number of retries. If this action does not complete successfully, repeat the same action on another domain. The sequence diagram for Scenario 2.a can be seen in Fig. 10, where N (number of retries) is set to be 1.

3. Other Job faults :

   a. The recovery actions for 1.a, 1.b and 1.c (*e.g.* Resubmit Job pattern) fail. In this case the job is redirected to a new domain.
   b. A "No Service" fault occurs when a job submission request is sent to a domain, but the service does not exist due to either the service was never deployed, or it was migrated. (For e.g. HTTP 404 error: Page Not Found). In this scenario, there is no benefit in retrying the job at the same domain, and hence can be directly redirected.

Proxy Action: Redirect the job submission request to another domain. Policies specify the number of maximum redirecting attempts value.

Table 2 summarizes results from our experiments that simulate the above mentioned failure and recovery scenarios. Each row reports the number of patterns applied by the Proxy in different fault scenarios and the average execution times observed across multiple runs of the Montage workflow. Test 1 represents the base case with no faults. For all the other cases, the same policy file was used with the *timeout = 20* seconds, *retryInterval = 20* seconds and *maxRetry* = 2. In order to measure the time values in isolation from the specific service that was faulted, each enumerated fault within Test 2-4 was generated at the same point (i.e. before the same service call for the specific task) throughout the workflow.

Total execution time values for Test 3 and Test 4 are comparable as the Proxy behavior is similar for the "Service Unavailable" and "Connection Error" fault types. However, Test 2 has a higher execution time than both Test 3 and Test 4, because while the faults are reactively and immediately raised, the timeouts are passively observed. Test 5 has much higher execution time than the remaining tests, substantially because of the large number of Redirects. In this test, upon a "Service Unavailable" or "Connection Error" fault, a job redirection occurs after the Retry Job pattern fails. This means that each extra Redirect introduces at least 40 seconds (maxRetry * retryInterval) to the total execution time.

Table 2. Runtime table for 5 different simulations of the Montage workflow. Number of "Retry Job", "Redirect" and "Retry Operation" patterns applied by the Proxy is given for each service and each fault type.

	Fault Type / Service (Pattern)	Timeout	Service Unavailable	Connection Error	No Service	Total Execution Time
Test 1	***SubmitJob*** (Retry Job/Redirect)	0/0	0/0	0/0	-/0	18 min. 46 sec.
	GetProperties (Retry Operation/Redirect)	0/0	0/0	0/0	-/0	
Test 2	***SubmitJob*** (Retry Job/Redirect)	3/1	0/0	0/0	-/0	23 min. 18 sec.
	GetProperties (Retry Operation/Redirect)	3/1	0/0	0/0	-/0	
Test 3	***SubmitJob*** (Retry Job/Redirect)	0/0	3/1	0/0	-/0	22 min. 06 sec.
	GetProperties (Retry Operation/Redirect)	0/0	3/1	0/0	-/0	
Test 4	***SubmitJob*** (Retry Job/Redirect)	0/0	0/0	3/1	-/0	21 min. 56 sec.
	GetProperties (Retry Operation/Redirect)	0/0	0/0	3/1	-/0	
Test 5	***SubmitJob*** (Retry Job/Redirect)	0/0	2/1	2/1	-/1	25 min 43 sec.
	GetProperties (Retry Operation/Redirect)	0/0	2/1	2/1	-/1	

7 Related Work

Condor [17] addresses faults that occur due to job failure, communications faults and other unusual and erroneous conditions via job resubmissions and restarts. DAGMan, the workflow engine in Condor, is responsible for enforcing the dependencies between the jobs defined in the workflow. In case of job failure, DAGMan can retry a job for a given number of times, or a job flow generated as a rescue DAG can be potentially modified and re-submitted at a later time.

The approach in BPEL4JOB [4] investigated the incorporation of fault handling policies in BPEL workflows at flow model development time. It used a two staged methodology: embedding only the recovery policies in the application flow and then expanding the application flow to include the recovery processes. Unlike our approach to handle faults at execution time, this approach would require modification of the application flow and modeling tooling.

Since modeling-time fault-handling requires knowledge of all faults a-priori, an alternate approach is to handle these failures at runtime. The Pegasus project [18] provides support for just-in-time planning, which follows a lazy approach for mapping sub-flows to the currently available resources. This approach works better than static mapping in dynamic resource conditions. However, if a job fails at runtime, it blindly re-schedules the entire sub-flow without looking into the source of the problem. In [19], the authors study the impact of runtime optimizations made at the scheduler for handling workload surges, while minimizing the reconfiguration overhead. However, identification of failure causes and fault-tolerant patterns is not addressed. Prior work in [9] characterizes workflow exceptions for web services into a set of patterns that specify how the individual task on which the exception is based and all other dependent tasks should be handled and what recovery action (if any) is to be undertaken. In the workflow domain, worklets [10] have been proposed to build extensible dynamic fault-handling services for Yet Another Workflow Language (YAWL). This work presents a detailed taxonomy of exception patterns in the web services domain from which a dynamic runtime selection is made depending on the context of the exception and the particular work instance.

8 Conclusion and Future Work

In this paper, we proposed the approach of addressing the fault tolerant issues at deployment and runtime, in comparison to various fault recovery strategies at the modeling time. We adapt job flows at deployment time and automatically incorporate context information to be used by the Transparent Proxy, which intercepts potential faults at runtime. The Transparent Proxy searches the populated knowledge-base with recurrent job flow patterns and fault tolerant patterns, and then finds the pre-defined recovery strategies from the recovery policies to handle the fault. This approach has two distinct advantages: (1) maintaining separation of concerns between application flow and recovery strategies; and (2) flexibility of defining job flow patterns, fault patterns, and recovery strategies as late as deployment or runtime. We validated our approach with a prototypical implementation using ActiveBPEL workflow engine, GridWay Meta-scheduler and the Montage application and presented experimental

data collected on the validation system. Our current experimentation uses a selected set of job flow and fault patterns, and a limited set of recovery strategies expressed simply in a policy file. Possible future work would include exploration of additional flow and fault patterns and more sophisticated recovery strategies using semantic and data information of the job flows.

Acknowledgement. This work was supported in part by IBM, the National Science Foundation (grants OISE-0730065, OCI-0636031, HRD-0833093, and HRD-0317692). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect those of the NSF and IBM.

References

1. Berriman, G.: Montage: A Grid enabled image mosaic service for the national virtual observatory. Astronomical Data Analysis Software and Systems XIII (2003)
2. Taylor, I.J., et al. (eds.): Workflows for e-Science. Springer, Heidelberg (2007)
3. Brown, G.D.: Z/OS JC, 5th edn. Wiley Publisher, Chichester (2002)
4. Tan, W., Fong, L., Bobroff, N.: BPEL4Job: a fault-handling design for job flow management. In: Krämer, B.J., Lin, K.-J., Narasimhan, P. (eds.) ICSOC 2007. LNCS, vol. 4749, pp. 27–42. Springer, Heidelberg (2007)
5. Ezenwoye, O., Sadjadi, S.M.: TRAP/BPEL: A Framework for Dynamic Adaptation of Composite Services. In: International Conference on Web Information Systems and Technologies (WEBIST-2007), Barcelona, Spain (2007)
6. Dasgupta, G., Ezenwoye, O., Fong, L., Kalayci, S., Sadjadi, S.M., Viswanathan, B.: Design of a Fault-Tolerant Job-Flow Manager for Grid Environments Using Standard Technologies, Job-Flow Patterns, and a Transparent Proxy. In: Proceedings of 20th International Conference on Software Engineering and Knowledge Engineering (SEKE), Redwood City, CA (July 2008)
7. ActiveBPEL, `http://www.activevos.com/community-open-source.php`
8. Huedo, E., Montero, R.S., Llorente, I.M.: The GridWay Framework for Adaptive Scheduling and Execution on Grids. In: Workshop on Adaptive Grid Middleware, Intl. Conf. Parallel Architectures and Compilation Techniques (PACT 2003) (September 2003)
9. Russell, N., van der Aalst, W.M.P., ter Hofstede, A.H.M.: Workflow Exception Patterns. In: Dubois, E., Pohl, K. (eds.) CAiSE 2006. LNCS, vol. 4001. Springer, Heidelberg (2006)
10. Adams, M., ter Hofstede, A.H.M., Edmond, D., van der Aalst, W.M.P.: Dynamic and Extensible Exception Handling for Workflows: A Service-Oriented Implementation. BPM Center Report BPM-07-03, BPMcenter.org (2007)
11. Jordan, D., et al.: Web Services Business Process Execution Language Version 2.0 (2007), `http://docs.oasis-open.org/wsbpel/2.0/CS01/wsbpel-v2.0-CS01.pdf`
12. IBM Websphere Process Server, `http://www-306.ibm.com/software/integration/wps/`
13. Anjomshoaa, A., et al.: Job Submission Description Language (JSDL) Specification v1.0. Proposed Recommendation from the JSDL Working Group (2005), `http://www.gridforum.org/documents/GFD.56.pdf`
14. Bobroff, N., Fong, L., Kalayci, S., Liu, Y., Martinez, J.C., Rodero, I., Sadjadi, S.M., Villegas, D.: Enabling Interoperability among Meta-Schedulers. In: IEEE 8th International Symposium on Cluster Computing and the Grid (ccGrid) (May 2008)

15. Foster, I., Kesselman, C.: Globus: A Metacomputing Infrastructure Toolkit. In: Proceedings of the Workshop on Environments and Tools for Parallel Scientific Computing, SIAM, Lyon, France (August 1996)
16. Rajic, H., et al.: Distributed Resource Management Application API Specification 1.0. Technical report, DRMAA. Working Group - The Global Grid Forum (2003)
17. Couvares, P., et al.: Workflow Management in Condor. In: Taylor, I.J., et al. (eds.) Workflows for e-Science. Springer Press, Heidelberg (2007)
18. Deelman, E., et al.: Pegasus: a Framework for Mapping Complex Scientific Workflows onto Distributed Systems. Scientific Programming Journal 13(3), 219–237 (2005)
19. Dasgupta, G., Dasgupta, K., Viswanathan, B.: Data-WISE: Efficient management of data-intensive workloads in scheduled Grid environments. In: Proceedings of IEEE/IFIP Network Operations and Management Symposium, NOMS (2008)