Editor: M. Brian Blake • mb7@cse.nd.edu

Managing Faults for Distributed Workflows over Grids

Onyeka Ezenwoye • *South Dakota State University*
M. Brian Blake • *University of Notre Dame*
Gargi Dasgupta • *IBM India Research Lab*
Liana L. Fong • *IBM T.J. Watson Research Center*
Selim Kalayci and S. Masoud Sadjadi • *Florida International University*

Grid applications composed of multiple, distributed jobs are common areas for applying Web-scale workflows. Workflows over grid infrastructures are inherently complicated due to the need to both functionally assure the entire process and coordinate the underlying tasks. Often, these applications are long-running, and fault tolerance becomes a significant concern. Transparency is a vital aspect to understanding fault tolerance in these environments.

The Internet's emergence and popularity has amplified the attention researchers and practitioners give to developing applications that execute across distributed, high-throughput, high-performance infrastructures. Grid environments allow applications from provider organizations to process information that might be provided by multiple consumer organizations. The key to this interaction is to take advantage of the computing resources local to all collaborating organizations. When new capabilities require the workflow integration of distributed applications, then a need exists for a workflow manager module that ensures a grid workflow application's functional process and a job scheduler module that actually enacts the operational code that realizes each underlying task. Manager and scheduler coordination requires support for both fault tolerance and recovery policies. Very often, scientists can't treat a job's failure within a flow in isolation, and they might need to apply recovery actions to preceding and dependent jobs as well. Grid services' interaction with dynamic distributed resources makes fault tolerance a critical and challenging aspect of workflow management. Fundamentally, two significant options exist for performing fault tolerance in grid workflow environments.

The first is failure-management logic weaved into the code. This approach adds the complexity of fault-recovery logic to the application flow. For example, Wei Tan and his colleagues investigated how to incorporate fault-handling and recovery strategies for jobs at modeling time without requiring the user to embed the recovery logic.[1] However, their work requires modifying the original flow to incorporate fault-handling policies at modeling time.

The second option is managing workflow failures at runtime in parallel with functional processing. As with TRAP/BPEL (Transparent Reflective Aspect Programming/Business Process Execution Language),[2] an intermediate proxy captures operational commands from the workflow engine. This approach uses a separate runtime failure-handling approach for stateless Web services. However, unlike stateless services, defining recovery for stateful grid jobs is more challenging because jobs can fail at different stages of execution and might require different recovery actions. Long-running jobs often require elaborate cleanup phases in the event of failure. Grid workflow developers, who often aren't computer scientists, have trouble accounting for all the service interaction's dynamics.

We believe that the most elegant approach

 1089-7801/10/$26.00 © 2010 IEEE

is a two-phased, semi-automated technique. The first phase, occurring at deployment time, consists of automatic functional assessment techniques that decompose workflow logic into a set of recurrent application behavioral patterns.[3] The second phase, during operation time, combines these behavioral patterns with user-defined policies to enable proxy services that transparently intercept and monitor job submissions and resolve faults at runtime. Here, we describe this approach and discuss the principles that make it effective.

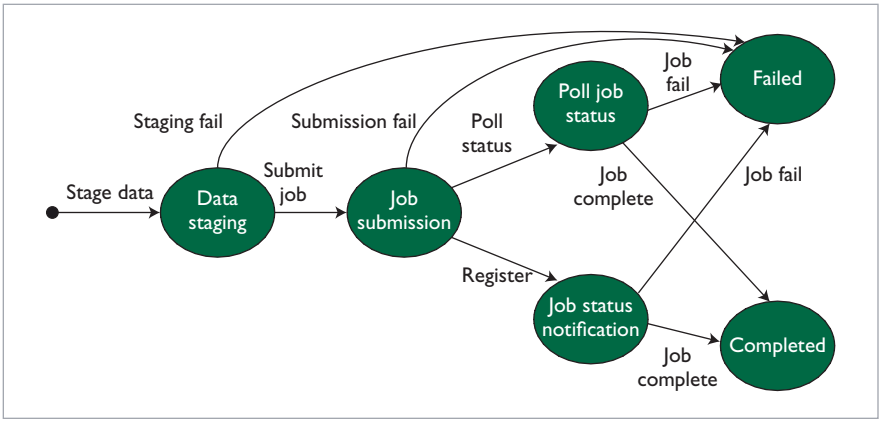

Figure 1. Statechart diagram capturing typical application behavioral patterns. We can see all the possible states in a job execution's life cycle. A failure in one or more of these activities entails a transition to the failed state; each such transition represents a fault pattern.

Understanding Fault Tolerance

Previous research has broadly categorized workflow failures as work item failures, deadline expiry, resource unavailability, external triggers, and constraint violation.[4] We can handle some failures well by specifying recovery actions at modeling time. However, in an uncontrolled grid environment, exceptions also occur. Handling all this at modeling time is infeasible due to the high complexity that it adds to the workflow. It's also infeasible to know a priori all the different failure scenarios that could arise. Several patterns exist that explain workflow and fault tolerance in these environments.[3]

Workflow Patterns

Several patterns capture reusable concepts in grid workflows:

- *Data staging*. Many grid jobs require input data, and, absent a shared file system, these data sets must be at the execution site (that is, staged in). A typical data-staging pattern in workflows comprises staging in data from either producer jobs or defined inputs, followed by a job submission pattern.
- *Job submission*. During synchronous job submission, the workflow application submits jobs for execution, and the submission call doesn't return until a job completes. However, during asynchronous submission, the job submission call returns with a job identifier with which job status updates can be obtained.
- *Job monitoring*. The workflow application can obtain job status updates via polling or notification mechanisms. The workflow captures job execution completion status during various states of workflow operation. Some job execution failures can be handled by either resubmitting the job at the same domain or by migration to a new domain.

We can broadly classify fault patterns observed due to workflow failure as job-submission failure, data-staging failure, job-execution failure, job-notification failure, and job-query failure. Figure 1 illustrates the interaction among these failures.

Fault-Tolerance Patterns

For the faults shown in Figure 1, we can also capture recovery actions as patterns (that is, fault-tolerance patterns):

- *Retry job*. The workflow application resubmits a job for execution when an exception occurs during job submission or execution. In this pattern, the workflow submits jobs to the same domain. Resubmitting the job might require modifications in the job specification and resource requirements. For example, the workflow can recover submission failures that arise from the scheduling service's temporary unavailability by resubmitting the job, whereas execution errors require a more detailed analysis of job state, status, and exit codes as well as significant domain expertise for fault handling. The workflow should automatically repoll or reregister for the new submission.
- *Redirect job*. Similar to the retry job pattern, the workflow redirects a job to a different domain for execution when an exception occurs during job submission or execution. The workflow might select this fault-tolerance pattern because all its previous attempts at resubmitting to the same domain have failed or because the workflow has decided that submitting the job to the current domain has a low probability of success. The workflow restages the redirected job's input data at the new target domain. Output data might need to be staged out.

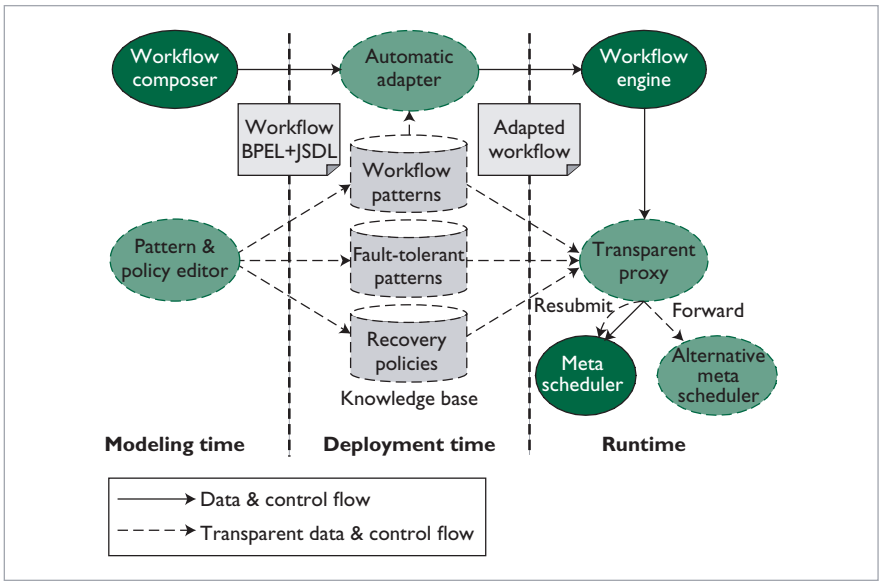

Figure 2. A fault-tolerant grid workflow architecture. Solid lines refer to components related to the workflow's core business logic; dashed lines relate to fault-tolerance and recovery components.

- *Retry query*. The workflow resumes polling for job-status registration for notification upon job resubmission. It transparently repolls the newly submitted job (possibly using the newly returned job ID), which involves translating and modifying the original polling messages from the flow manager to map to the newly resubmitted job's repolling.
- *Force-fail*. When no further progress is possible, the workflow changes the job state to "failed".

These workflow and fault-tolerance patterns are reusable concepts that we've used in the design of a fault-tolerant workflow manager that can handle failures at runtime.

Supporting Transparent Fault Tolerance

Figure 2 illustrates an architecture for supporting fault tolerance in grid workflow environments. The components and arrows with solid lines indicate parts of the architecture related to the workflow's core business logic; those with dashed lines relate to the workflow's fault-tolerance and recovery portions. Within this architecture, domain experts use the *workflow composer* to specify the application's business logic using Oasis's Web Service BPEL (WS-BPEL)[5] and the Global Grid Foundation's Job Submission Description Language (JSDL),[6] which we refer to here as BPEL+JSDL. The domain expert should be concerned only with the application's business logic and not with handling faults and exceptions.

At deployment time, the developer passes the resulting workflow to the *automatic adapter*, which in turn automatically generates a functionally equivalent workflow with context information. The *proxy* uses this embellished workflow description to monitor the interaction between the flow manager and the *metaschedulers*. The automatic adapter leverages an algorithm that identifies the known application behavioral patterns (such as job submission) within the workflow. The *knowledge base* stores the most updated application behavioral patterns. The developer can add new behavioral patterns to the knowledge base using the *pattern and policy editor*. The generated workflow, called the *adapted workflow*, would include context information required to coordinate job execution. However, the adapted workflow doesn't have any knowledge of how to handle faults at runtime. Instead, the adaptation incorporates some generic code for interception at sensitive points in the original BPEL+JSDL workflow. The most appropriate place to insert interception code in such a workflow is at the interaction points (that is, at the scheduler service invocation). The inserted code takes the form of standard BPEL constructs that automatically replace invocations to the scheduler with invocations to the proxy. Using standard BPEL constructs ensures that the modified process is portable. This adaptation lets the *transparent proxy* modify the BPEL+JSDL workflow behavior at runtime.

Also at runtime, the *workflow engine* will execute the BPEL+JSDL workflow. A workflow engine can be any BPEL engine without modification or extension. Note that during the automatic workflow adaptation, all the calls originally targeted for the local metascheduler are redirected to the transparent proxy. So, the transparent proxy will intercept all calls to the metascheduler. The proxy will appear as a metascheduler to the workflow process and as a workflow process to the metascheduler — hence the name "transparent." Its main responsibility includes submitting the intercepted jobs to the local metascheduler and notifying the workflow process of the job status when it receives job-status updates from the metascheduler. In addition, it implements a pattern-matching algorithm that monitors the intercepted calls' behavior and provides fault-tolerant behavior when faults occur.

We've implemented this architecture in a fault-tolerant workflow management prototype. The prototype comprises a workflow manager, proxy, and scheduler component.

To build this testbed, we used the ActiveBPEL flow engine (v4.1; www.activevos.com/community-open-source.php), a metascheduler based on the Globus Toolkit (GT4),[7] and the GridWay metascheduler (v5.2.3). We used the Distributed Resource Management Application API (DRMAA)[8] for job submission, monitoring, and control.

Effective Fault-Tolerance Principles

Based on the previously mentioned architecture, fault tolerance in grid workflow environments requires several principles to be effective:

- *Generalized workflow behavior.* Transparency requires the flexibility to define application behavioral patterns, fault-tolerant patterns, and recovery policies, which might be stored in the system's knowledge base as late as deployment and runtime.
- *Behavioral pattern evolution.* Recovery policies' sophistication can grow with the knowledge of flow and fault patterns. Because workflows in the same application domain are functionally similar, developers must capture new common behavior as abstract, reusable concepts that are reconfigurable. Captured domain knowledge can be used to model the specification of workflows, and application generators used to produce new systems from those high-level abstractions.
- *Separation of workflow functionality and resources.* The workflow manager is responsible for executing the workflow according to the specified logic while being agnostic to resource allocation decisions. This allows for system scalability for functional and nonfunctional concerns.

Effective fault-tolerance principles must align with the emerging technologies and techniques for deploying next-generation grid infrastructures.

Future Challenges

As societal problems increase in scope, specialized solution architectures are needed to encapsulate multiple applications distributed remotely across multi-organizational resources. New challenges occur in handling faults in these next-generation environments.

Multitier distributed application architectures (for example, applications with a Web portal or an application and a database layer) are difficult to debug because faults could be spread across tiers. Each tier typically has its own failure footprint. So, instead of a generic proxy, specialized proxies for each tier with specialized tier-specific fault patterns or recovery policies might be necessary.

Technologies such as cloud computing are becoming hugely popular because of their cost advantages. However, managing such computing environments is complicated by an inherent lack of centralized control. Fault management in these shared infrastructures might need multiple proxies across resource domains to collaboratively detect faults and take corrective action.

On some occasions, failures in one application component can trigger failures in others. Cascading faults can often hide the primary problem in a multicomponent distributed application. Accurate fault-detection tools exist that analyze all components' failure logs to reconstruct the failure history and identify fault. However, as we distribute applications across domains, such fault history reconstruction becomes impossible. New tools and technologies are needed to deal with such situations. A significant challenge is how to model such distributed systems' state quickly and accurately.

We need efficient algorithms that can automatically adapt complex workflows before deployment and during execution to transparently enhance workflow execution. The algorithms should transform the directed graphs to support dynamic adaptation and optimal execution. Novel techniques are required that automatically modify existing workflow definitions to address concerns such as efficient data staging or redundancy introduction, in order to make them reconfigurable at runtime. This process simplifies the composite model specification, which would otherwise be cumbersome for domain specialists.

Providing fault tolerance for grid workflows requires end-to-end coordination between the workflow manager and job scheduler. Robust workflow management for grid environments must separate application business logic and recovery strategies while providing flexibility for defining application behavioral patterns, fault patterns, and recovery strategies late in the workflow life cycle.

References

1. W. Tan, L. Fong, and N. Bobroff, "BPEL-4Job: A Fault-Handling Design for Job Flow Management," *Proc. 5th Int'l Conf. Service-Oriented Computing*, Springer, 2007, pp. 27–42.
2. O. Ezenwoye and S.M. Sadjadi, "TRAP/BPEL: A Framework for Dynamic Adaptation of Composite Services," *Proc. Int'l Conf. Web Information Systems and Technologies*, Springer, 2007.
3. G. Dasgupta et al., "Design of a Fault-Tolerant Job-Flow Manager for Grid Environments Using Standard Technologies, Job-Flow Patterns, and a Transparent Proxy," *Proc. 20th Int'l Conf. Software Engineering and Knowledge Engineering*, Knowledge Systems Inst., 2008, pp. 814–819.
4. N. Russell, W.M.P. van der Aalst, and A.H.M. ter Hofstede, "Workflow Exception Patterns," *Proc. 18th Int'l Conf.*

Advanced Information Systems Engineering, E. Dubois and K. Pohl, eds., LNCS 4001, Springer, 2006, pp. 288–302.
5. D. Jordan, *Web Services Business Process Execution Language Version 2.0.*, Oasis, 2007; http://docs.oasis-open.org/wsbpel/2.0/CS01/wsbpel-v2.0-CS01.pdf.
6. A. Anjomshoaa et al., *Job Submission Description Language (JSDL) Specification v1.0*, JSDL Working Group, 2005; www.gridforum.org/documents/GFD.56.pdf.
7. I. Foster and C. Kesselman, "Globus: A Metacomputing Infrastructure Toolkit," *Int'l J. Supercomputer Applications and High Performance Computing*, vol. 11, no. 2, 1997, pp. 115–128.
8. H. Rajic et al., *Distributed Resource Management Application API Specification 1.0.*, tech. report, DRMAA (Distributed Resource Management Application API) working group, Global Grid Forum, 2003.

Onyeka Ezenwoye is an assistant professor at South Dakota State University. His research interests include adaptive service composition, fault-tolerant distributed systems and software engineering. Ezenwoye has a PhD in computer science from Florida International University. He's a member of the IEEE and the ACM. Contact him at onyeka.ezenwoye@sdstate.edu.

M. Brian Blake is a professor in the Department of Computer Science and Engineering at the University of Notre Dame. He conducts applied research in the areas of software engineering, service-oriented computing, and enterprise integration in collaboration with various government and industry organizations. Blake has a PhD in information and software engineering from George Mason University. He's a senior member of the IEEE. Contact him at m.brian.blake@nd.edu.

Gargi Dasgupta is a research staff member at IBM Research, India. Her research interests include power and energy management solutions for servers and systems, new computing models like the cloud and virtualization, and other aspects of distributed systems management. Dasgupta has a PhD in computer science from the University of Maryland. Contact her at gdasgupt@in.ibm.com.

S. Masoud Sadjadi is an assistant professor at the School of Computing and Information Sciences at Florida International University. His research interests include distributed systems, software engineering, autonomic computing, high-performance computing, grid computing, and cloud computing. Sadjadi has a PhD in computer science from Michigan State University. He can be reached at sadjadi@cs.fiu.edu.

Selim Kalayci is a PhD candidate at the School of Computing and Information Sciences at Florida International University. His research interests include distributed systems, service-oriented computing, autonomic computing, grid computing, and cloud computing. He's a member of the IEEE and the ACM. Contact him at skala001@cs.fiu.edu.

Liana L. Fong is a research staff member and manager at IBM T.J. Watson Research Center. Her research interests include operating system structures, resource and system management, and performance evaluation for both high-performance computing and commerical workloads on large-scale distributed systems. She's a senior member of the ACM. Contact her at llfong@us.ibm.com.

Selected CS articles and columns are also available for free at http://ComputingNow.computer.org.