ABSTRACT

MOUALLEM, PIERRE A. A Fault Tolerance Framework for Kepler-based Distributed Scientific Workflows. (Under the direction of Dr. Mladen A Vouk).

Scientific Workflow Management Systems (S-WFMS), such as Kepler, have proven to be an important tool in scientific problem solving. They allow scientific experiments to handle increased concurrent and longitudinal complexity and larger amounts of data. Since S-WFMS rely on a multitude of technologies and principles, they share the vulnerabilities and errors of these technologies, and since they hide the underlying components, identifying those errors becomes a challenge, especially when operating in a tightly coupled environment.

Fault tolerance and failure recovery in scientific workflows running in S-WFMS is still a relatively a young topic. The work done in the domain mostly applies classic fault tolerance mechanisms, such as Alternative Versions and Check-pointing. Often scientific workflow systems simply rely on the fault tolerance capabilities provided by their third party subcomponents such as schedulers, Grid resources, or the underlying operating systems. When failures occur at the underlying layers, a workflow system sees this as failed steps in the process, often without additional detail. Therefore, the ability of the system to recover from those failures may be limited.

The approach described in this dissertation aims to provide a light weight end-to-end framework implementing fault tolerance in situations where Kepler is the workflow engine. Based on failure patterns that occur in real-life scientific workflow executions, we present methodologies which inject recovery fragments into fault-prone scientific workflow models to achieve end-to-end fault tolerance. The framework offers mechanisms for capturing the

failures at different layers of a workflow management architecture. We discuss the framework in the context of the failures observed in two production-level Kepler-based scientific workflows, specifically XGC and S3D. The framework is divided into three major components:

(i) A general contingency Kepler actor that provides a recovery block functionality at the workflow level. This actor extends the concept of Alternative versions (also known as Recovery Block) by adding a provenance based scoring scheme to the decision making process.

(ii) An external monitoring module that tracks the underlying workflow components, monitors the overall health of the workflow execution, and communicates with the workflow engine and the contingency actor to better address those errors, and

(iii) A checkpointing mechanism which handles loops, stateless and stateful actor. The mechanism provides “smart resume” capabilities for cases in which an unrecoverable error occurs.

This framework takes advantage of the provenance data collected by the Kepler-based workflows to detect failures and help in fault tolerance decision making. The three components are interconnected via sockets and database calls.

Capabilities and limitations of the framework are assessed using simulations. The results show that the presented fault tolerance framework is quite cost effective. Framework operations incur a marginal overhead, dramatically increase the workflow reliability and at the same time reduce the average workflow runtime per successful run.

© Copyright 2011 by Pierre A Mouallem
All Rights Reserved

A Fault Tolerance Framework for Kepler-based Distributed Scientific Workflows

by
Pierre A Mouallem

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Computer Science

Raleigh, North Carolina

2011

APPROVED BY:

_____	_____
Dr. Nagiza Samatova	Dr. Yannis Viniotis
_____	_____
Dr. Mladen Vouk Committee Chair	Dr. Laurie Williams

DEDICATION

I dedicate this dissertation to my parents and my brother. I could never have done it without their constant support, encouragement and faith in me. Thank you for everything you have done, I am forever indebted.

BIOGRAPHY

Pierre Mouallem completed his primary education in Lebanon. He received his Bachelor's degree in Computer Science from Notre Dame University, Lebanon. He received his Master's degree in Computer Science from NC State University in 2005 and joined the PhD program in 08/2005.

ACKNOWLEDGMENTS

I express my sincere gratitude towards my advisor, Dr. Mladen Vouk, for his guidance, relentless support and encouragement. I also would like to thank Dr. Nagiza Samatova, Dr. Yannis Viniotis and Dr. Laurie Williams for serving on my committee and for their insight and assistance.

I would also like to thank my colleagues in the Scientific Process Automation group (SPA) of the DOE Scientific Data Management Center (SDM) and the SDM group for their support and help throughout the research. Special thanks to Roselyne Barreto, Norbert Podhorszki and Scott Klasky from ORNL, Ilkay Altintas and Daniel Crawl of SDSC, Bertram Ludaescher of UC-Davis, Ayla Khan of the University of Utah and last but not least, Meiyappan Nagappan of NCSU.

This work has been supported in part by the DOE SciDAC grant DE-FC02-07-ER25484 and IBM Shared University Program.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

API	Application Programming Interface
AVS	Advanced Visual Systems
DB	Database
DDF	Dynamic Data Flow
DOE	U.S. Department of Energy
FT	Fault Tolerance
HPC	High Performance Computing
LANL	Los Alamos National Laboratory
NERSC	National Energy Research Scientific Computing Center
ORNL	Oak Ridge National Laboratory
OS	Operating System
PN	Process Networks
RHEL	Red Hat Enterprise Linux
SCIDAC	Scientific Discovery through Advanced Computing
SDF	Synchronous Data Flow
SDM	Scientific Data Management
SNMP	Simple Network Management Protocol
SPA	Scientific Process Automation
SQL	Structured Query Language
SSH	Secure Shell
S-WFMS	Scientific Workflow Management System
VCL	Virtual Computing Laboratory
XGC	Gyrokinetic Particle-in-cell Code
XML	Extensible Markup Language

Chapter 1

Introduction

1.1 Scientific Workflows and their role in the Scientific Discovery

Comprehensive, end-to-end, data and workflow management solutions are needed to handle the increasing complexity of processes and data volumes associated with modern scientific problem solving. Often such workflows operate in networked environments. The key to the solution is an integrated network-based framework that is functional, dependable, and supports data and process provenance. Such a framework needs to make development and use of application workflows dramatically easier so that scientists' efforts can shift away from data management and utility software development to scientific research and discovery.

The Scientific Process Automation group (SPA) [1] of the DOE Scientific Data Management (SDM) Center [1] was formed to develop information technologies that would enable the scientific community better address managing the complexity and volume of data. As part of its charter, SDM center is researching, developing and deploying data management tools that address such challenges using the Kepler workflow management system [2]. Kepler is an open-source system on the PTOLEMY II framework [3].

A scientific workflow is defined as a set of interrelated structured activities and computations that represents or models a scientific experiment. Automated implementation of such as model allows scientists to better manage, visualize and analyze, and modify their experiments. Scientific workflow includes actions, decisions (control-flows), information flows (data-flow), exception and interrupt handling (e.g., event-flows), and the underlying coordination and scheduling required to execute a workflow (orchestration). In its simplest case, a workflow is a linear sequence of tasks.

Kepler workflows often operate on the "control plane." Figure 1.1 illustrates this.

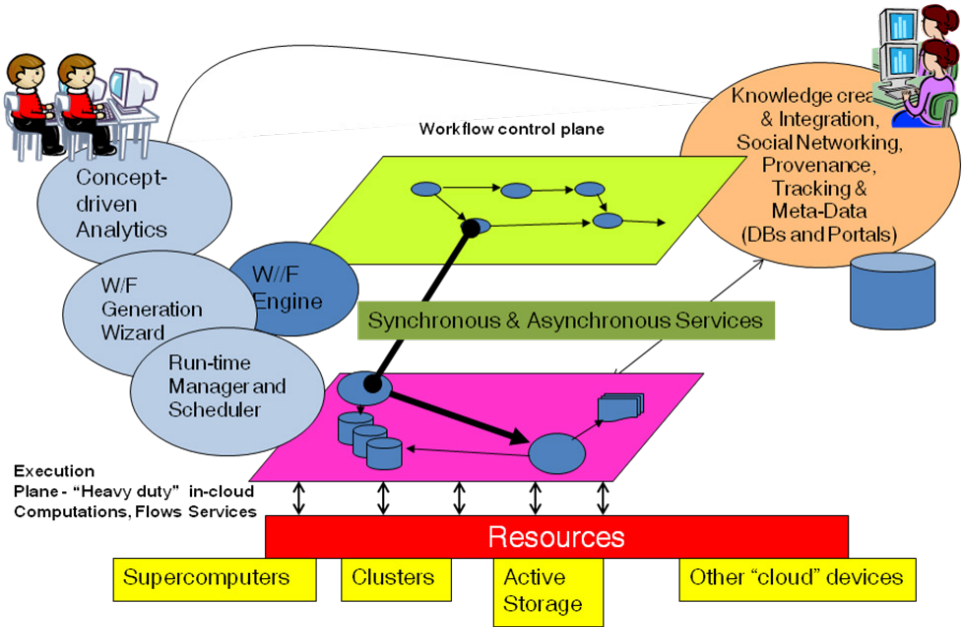

Figure 1.1: Illustration of the control plane orchestration of workflows and resources.

In a Kepler workflow, a process is called an actor. Actors are interconnected by communication channels through which the data flow in the form of tokens. A convenient

representation is offered by a graph. Execution of the whole workflow can have two parts – execution on the control plane (Kepler, Figure 1.1) where actors act as flow routers and intermediaries that manage real resources on supercomputers, dedicated clusters and in the cloud. Such a workflow is controlled by one of a number of special schedulers called Directors. SDM has also developed a provenance collection framework for Kepler [4], along with visual interfaces to display and analyze the results – the dashboard [5]. This is illustrated in Figure 1.2

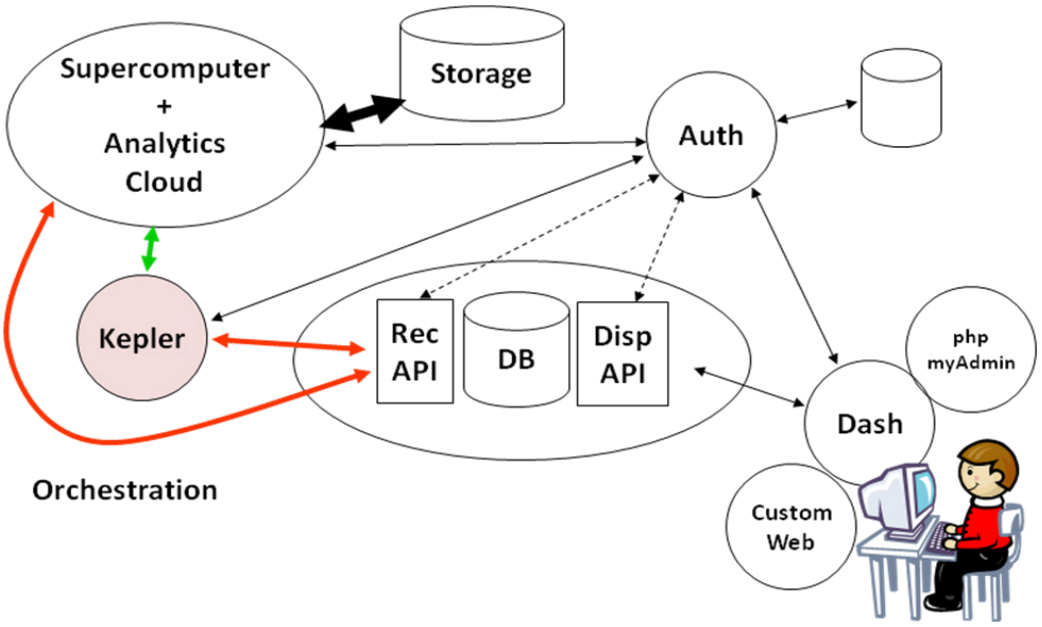

Figure 1.2: SDM framework that support provenance data collection, workflow management through a dashboard, and a Kepler orchestration engine.

1.2 Challenges facing Scientific Workflows

Scientific Workflow Management Systems (S-WFMS) have proven to be a valuable tool in designing and managing and steering scientific simulations as well as integrating data

analysis and visualizations [6]. As workflows become more complex, they also become more vulnerable to internal and external failures requiring fault tolerance during workflow execution [7] [8].

While classical fault tolerance solutions can operate well in the context of modern scientific workflows, some limitations arise from the nature of the scientific workflow engines, characteristics of workflow models, and the failure rates and bounds that application domain is willing to tolerate. Scientific workflows may include long-running, loosely coupled and repetitive computational steps that require the tolerance of failures during run-time with minimal impact on the overall execution. Fault tolerance approaches to scientific workflows should provide run-time failure avoidance and/or mitigation within workflow models and at different levels of an execution infrastructure. They also should be able to adapt to workflow evolution and to changes in its operational environments.

1.3 Suggested solution and Dissertation outline

In this dissertation, we examine the fault tolerance (FT) challenges observed for some of the high-end scientific workflows studied by the Scientific Process Automation group (SPA) of the Department of Energy (DOE) Scientific Data Management Center (SDM). The workflows, and FT framework we are discussing, are implemented using the Kepler workflow management system [2] as the orchestration tool. The framework leverages Kepler fault tolerance elements, and additional elements that compensate for control-plane blindness – inability of control plane (Figure 1.1) to recognize some of the lower level failure details.

The ultimate goal of the FT framework is to provide an appropriate end-to-end support for detecting and recovering from failures during execution of scientific workflows.

The remainder of the dissertation is organized as follows. In Section 2, we discuss the failure rates in High Performance Computing systems, and failure rates in Scientific Workflows. We also present two use cases of production level workflows, and discuss their vulnerabilities and failure rates. In Section 3, we present a survey of fault tolerance mechanisms in scientific workflows, their error detection and recovery capabilities, and we classify them based on the techniques used. In Section 4, we present a brief overview of the Kepler Provenance Framework, and the role it plays in enabling the different fault tolerance mechanisms. In Section 5, we present the first component of the fault tolerance framework developed in the work reported here, namely the "Forward Recovery" mechanism. We discuss the motivation, contribution and architecture details of this solution. In Section 6, we present the second component of our fault tolerance framework, namely the "Checkpointing" mechanism. We discuss the motivation, contribution and architecture details of this solution. In Section 7, we present the third component of our fault tolerance framework, namely the "Error-state handling layer". We discuss the motivation, contribution and architecture details of this solution. In Section 8, we present a use case and discuss the performance evaluation of the presented solution using emulation. Section 9 concludes and discusses future work.

1.4 Publications

A considerable amount of work discussed in this dissertation has been published by the author of the dissertation in the form of papers, tutorials and posters. For example, fault tolerance as it relates to scientific workflows is discussed in [9] [10] [11] [12]. Provenance management is scientific workflows is discussed in [13] [14]. Dashboard and simulation monitoring are discussed in [15] [16] [17]. Scientific workflows and their role in the scientific discovery process are discussed and presented in [18] [19] [20] [21].

Chapter 2

Failure Rate Analysis

2.1 Definition of failure

In order to understand the problem, let us start by formally defining the meaning of "failure" [22].

> "An **error,** or a mistake, made by a human or an external entity results in a **fault** in a software or system artifact[1] (e.g., missing or extra instructions in the code, wrong input settings, requirements or other documentation omissions, etc.). This fault can propagate through subsequent software stages and artifacts as one or more **defects**. When software is executed and encounters such a defect, it may go into an **error-state**. If an error-state results in an observed departure of the external result of software operation from software requirements or user expectations, we say that we have observed a **failure** of the software/system. Failures also can be caused by error-states that arise from lacks in user's knowledge and training (**how-to-use-errors)** and through **resource exhaustion** (e.g., out of disk space exceptions, hardware failure) that may or may not be properly handled by the software or the system. Failures and their frequency, tend to relate very strongly to customer satisfaction and perception of the product quality. Faults, on the other hand, are

[1] In the case of security attacks injection of faults is deliberate.

more developer oriented, since they tend to be translated into the amount of effort that may be needed to repair and maintain a system. Fault tolerance mechanisms try to prevent, or at least minimize, adverse impacts of failures. They can do that in a number of ways through forward recovery (e.g., masking of error-states, redundancy with voting mechanisms), and backward recovery (e.g., checkpointing, state recovery and re-execution)".

2.2 Failure rates in HPC systems

High performance computing systems, especially supercomputers, such as the ones available at Oak Ridge national laboratory [15], provide an invaluable tool for scientific research and discovery. However those systems are very expensive to own and maintain, and failures related to those systems can be very costly. Therefore studying and analyzing failures in such systems can improve reliability, and thus productivity, by better resource allocation and management [24].

There are several studies of failures in high performance computing systems, e.g., [25] and [26]. However, they are based on a few months of data at most, covering a small number of failures, and are relatively outdated. In 2005, Los Alamos National Lab released the raw data on failures that has been collected over a period of nine years [27], and in 2006, NERSC released the error data on I/O related failures collected over a period of 5 years [28]. Several studies were done to analyze that error data, such as [29] and [30].

These studies concluded that failure rates vary widely across systems, ranging from 20 failures to more than a 1000 failures per year, depending mostly on the system size (for example, systems with 4096 processors averaged 3 failures per day). Furthermore, they find that failure rates are linearly proportional to the number of processors per system.

Looking at the root causes of the failures, we noticed that around 50% were hardware related, 20% user related, 20% had unknown origin, and the remaining 10% split up among network, software and environment. As for the time to repair, the study finds that it varies between less than 3 hours and more than 10 hours, based on the type of error, with a mean time to repair of 355 minutes.

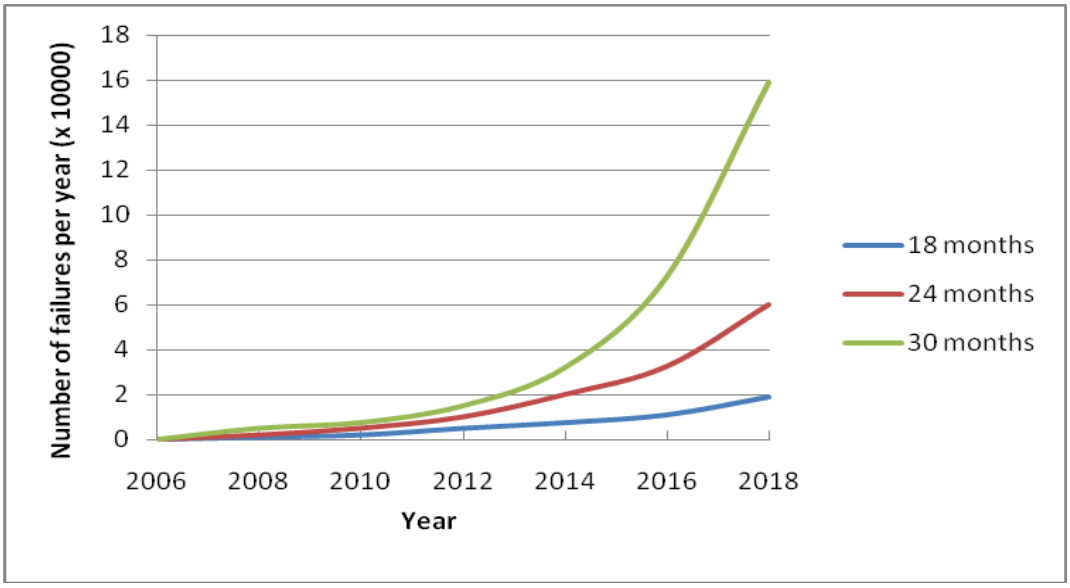

Figure 2.1: Expected growth in failure rate of multicore/multiprocessor systems

Figure 2.1, based on data from [29], shows the expected growth in failure rate as the size and complexity of supercomputers increases, and assuming that the number of cores will increase by a factor of 2 every 18, 24, or 30 months respectively. As the number of devices (processors, disks) increases, assuming no additional built in recovery mechanisms, so does the likelihood that the system is out of service for one reason or another. Furthermore, as the cost of those systems increases, those failures become very expensive. Therefore failure handling and reliability play a vital role in making systems scalable.

2.3 Failure rates in scientific workflow management systems

Both tightly coupled and loosely coupled systems become more failure prone as their size and complexity increases. Some systems, such as those found at ORNL, LANL and NERSC, are expensive, high priority, tightly controlled and closely maintained systems. They benefit from such additional close scrutiny. On the other hand, Grid based computational systems, such as the TeraGrid [31] and more recently cloud-based systems, may be more susceptible to failures due to the diverse and general-purpose nature of such infrastructures. The heterogeneous computing systems that make such grids, which are often loosely coupled by grid middleware, are much harder to monitor and control and therefore mitigate its failures.

Failure rates among grid based workflows vary greatly. This is due to factors such as the size and type of the workflow, the type and amount of resources it consumes, and the duration of the run.

For example, the authors in [32] describe the workflow system LEAD (Linked Environment for Atmospheric Discovery) which is used to study weather forecasting. Over a period of a month, 165 workflow runs were executed, with a total of 865 workflow steps [33]. Out of the 165 workflows 131 failed, due to a multitude of reasons mostly relating to errors that occurred in the grid infrastructure. That is a 78.8% failure rate, which meant that failures are the norm rather than exception. The authors later present a retry mechanism for their workflows which reduced the failure rate to approximately 23%, which is a large improvement; however that is still a large failure rate.

Another example is the work reported in [34], in which the authors studied the reliability of two grid platforms, TeraGrid [31] and Geon [35] via a deployment of a monitoring and benchmarking infrastructure. They conclude that the failure rates vary between 20% and 45%, based on the type and length of the test used.

2.4 Failure rates in Kepler based scientific workflows

2.4.1 Production level workflows use cases: XGC and S3D

In this section we discuss two use cases of production workflows that are used at Oak Ridge National Laboratory, and Sandia National Laboratory. We also discuss some use cases to describe some of the most commonly encountered errors.

XGC [36] [37] and S3D [38] are numerical simulation codes that scientists at the ORNL and Sandia national laboratory use to study fusion plasma effects and combustion phenomena,

respectively. Kepler-based workflows, e.g. [39], have been developed for both of these codes to automate the processing steps involved in deploying these codes on a supercomputer and analyzing the outcomes of the runs.

The XGC monitoring workflow is shown in figure 2.2 below, and the S3D workflow is shown in figure 2.3. These workflows might appear to have parallel execution flows, but effectively it is a serial flow, since the execution eventually blocks waiting on all the previous actor to finish before further proceeding.

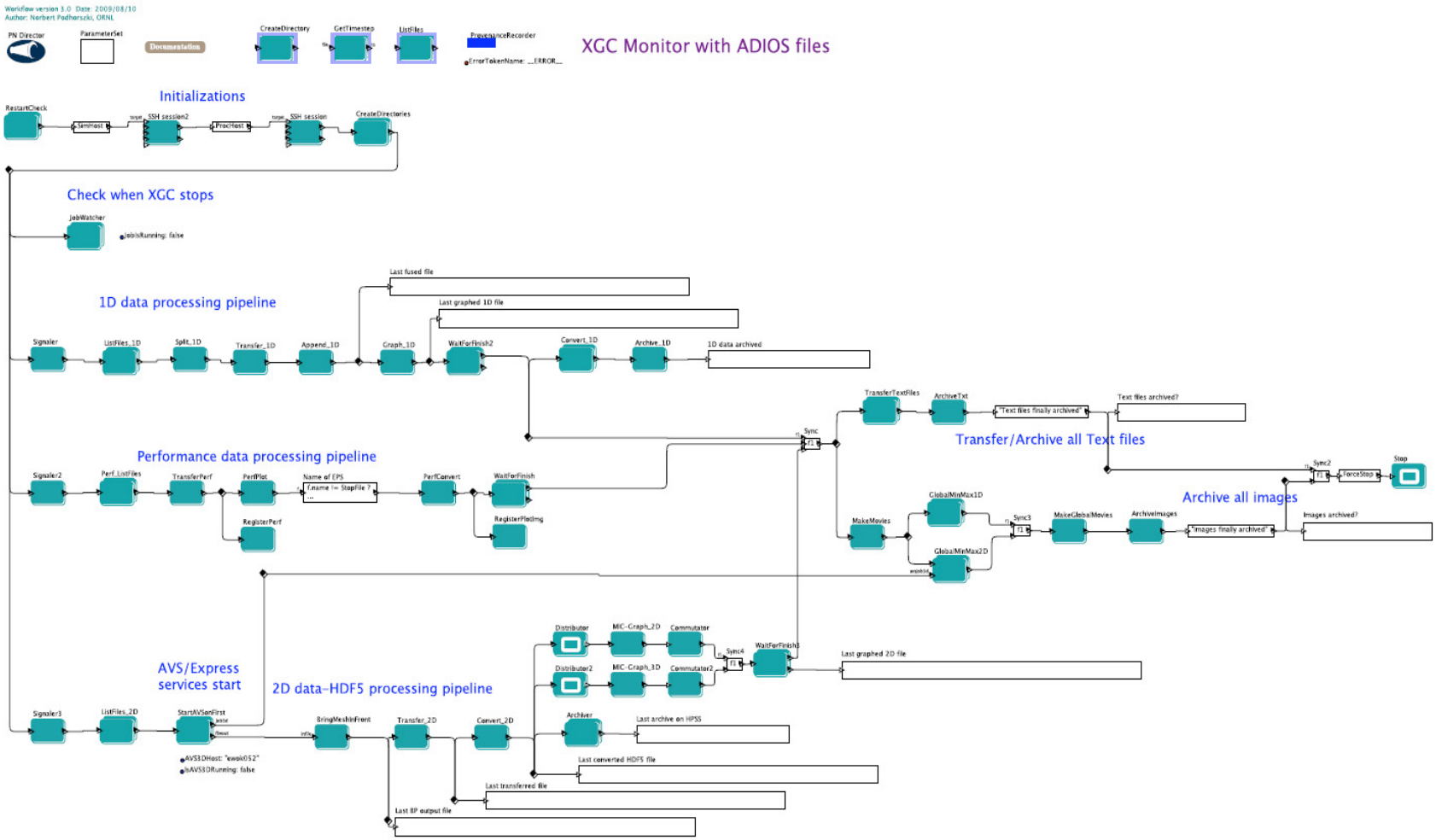

Figure 2.2: XGC Workflow

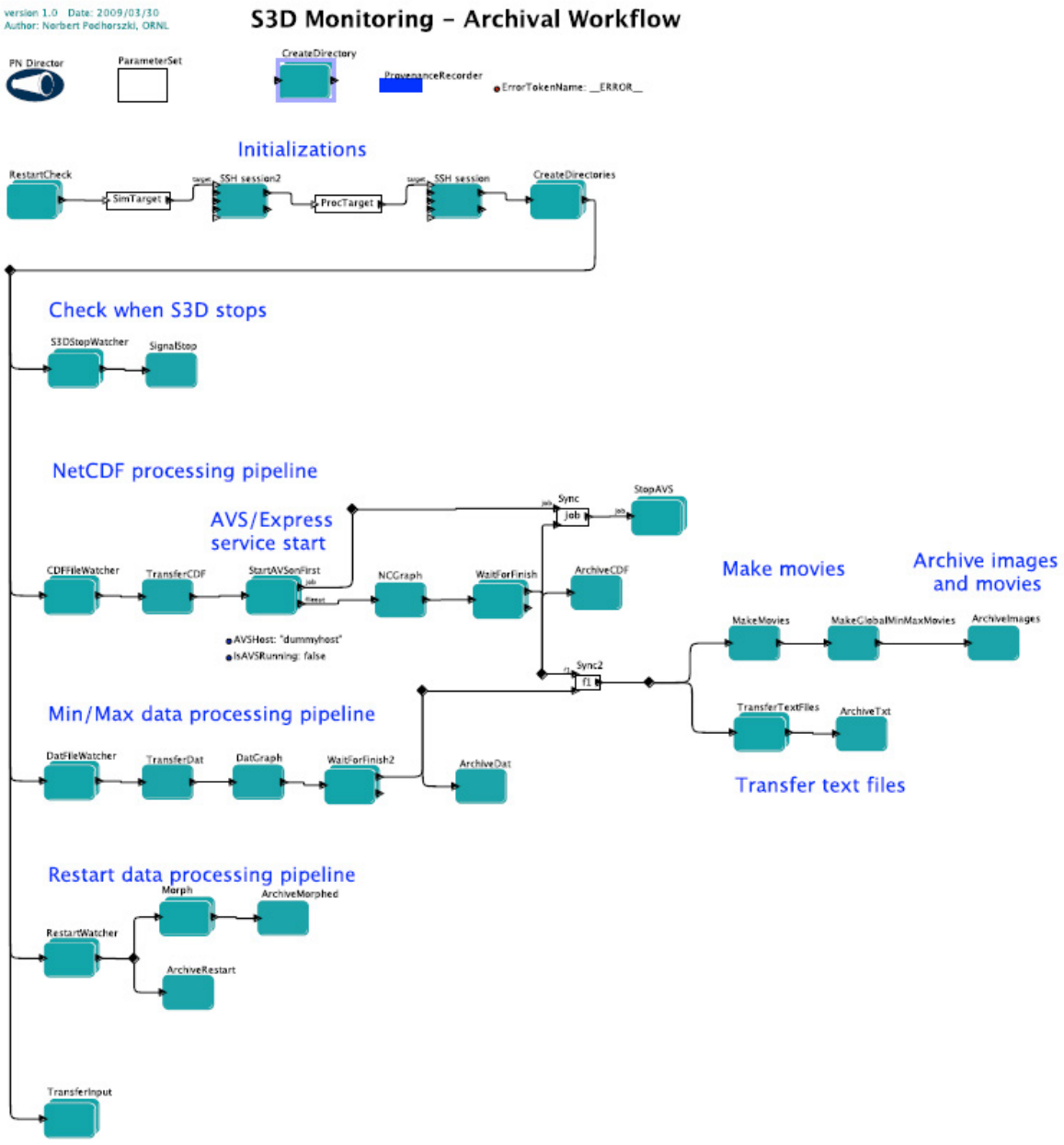

Figure 2.3: S3D Workflow

2.4.2 Typical failure scenarios

Both workflows are part of a family of workflows, that involve similar steps, known as deployment and monitoring workflows [40]. After computational codes have been launched on a supercomputer, these workflows monitor and manage outputs from long-running computations. Tasks include movement of data files to an analysis computer, archiving of the data, and generation of visualizations based on the progress of the computations. Due to their similarity, these workflows also exhibit similar categories of failures. Currently, both workflows also have an ability to effect some recovery from a failure through a light-weight re-start option – a feature that keeps track of successful task executions and does not re-execute them on re-start unless it is required by the task pipeline [41]. Below, we provide scenarios that highlight possible failures within these scientific workflows.

Scenario 1: Scientific workflows commonly access a diverse set of resources such as web services, databases, and external applications. A maximum time limit is often set when accessing these resources since they, or the networking link to them, may fail or become unresponsive. *When this limit is exceeded and a* ***timeout*** *occurs, how should workflow execution proceed?*

For example, the XGC workflow executes several applications to generate visualizations. When a timeout occurs in that part of the workflow, the workflow considers that this step has been completed "*successfully*," but no images or movies are produced. This may not always be a desirable way to handle such a failure, but it is probably the most expedient one in this case, since usually human intervention is needed to ascertain the cause of the problem.

Scenario 2: Kepler workflows often have long chains of actors linked together forming processing pipelines. *If a failure occurs while executing one of these tasks, what actions, if any, should be performed for the remaining tasks?*

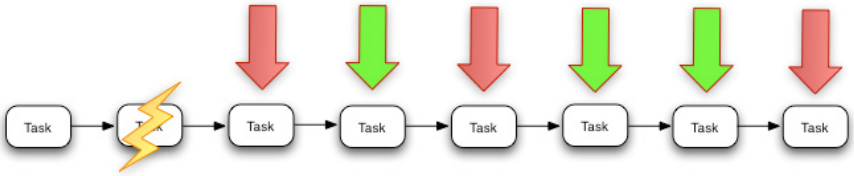

Figure 2.4: Failure in Processing Pipeline [12]

This situation is illustrated in Figure 2.4. It can occur in the S3D workflow [38], and the resulting actions depend on the state of the actor that failed and on downstream actors. If there is a failure in the second task, some of the subsequent activities should not be executed (denoted by the red arrows) as they can crash the overall workflow. However, other tasks (denoted with the green arrows) *should* be executed since their outcomes are necessary for the remainder of the workflow. This is currently a difficult design to implement as a workflow since the green activities cannot be gathered in an alternative path in a systematic manner due to inconsistencies of input/output types or their structuring over different paths. In this case, the fault tolerance takes on the challenge of structuring the tasks on conditional paths for every possible failure, input/output match and consistent precedence order. This means that the complexity is almost the same as building another workflow.

Scenario 3: Scientific workflows rely on a multitude of middleware services to successfully run a simulation. For example, the XGC workflow uses services for data manipulation, data transfer, visualizations, etc. These services are not a part of the workflow and therefore, are not monitored or controlled by the workflow; they are simply invoked from the workflow layer. *What happens if one of these services becomes unresponsive and the workflow layer does not realize it until it is too late, or not at all?* In either case the normal execution of the workflow is disrupted. *How does one handle that?* In the current workflow implementation, that would lead to a failed run, which would have to be restarted after the workflow scientist addresses the underlying problem. That is very inefficient since those run can take hours to complete, and restarting a run after it fails is time and resource consuming.

2.4.3 Fault classification

Analysis of workflow failures shows the range of issues that fault tolerance mechanisms need to handle in our case. Based on a logs of more than 1000 production runs of XGC and S3D workflows at ORNL collected between February 2009 and April 2010, (refer to Appendix A for sample production run log files), the failures and error-states encountered are summarized in table 2.1. Note that many failures are transparent to the workflow, and thus do not interrupt normal workflow execution. Unfortunately, in those cases, even when the workflow finished successfully the underlying simulation results were invalid.

Problems listed in table 2.1 can be classified based on which layer of the execution environment they originate. Figure 2.5 depicts three execution layers: the *Workflow Layer* provides control, directs execution, and tracks the progression of the simulation; the

Middleware Layer provides all services required for executing the simulation; and the *Hardware/OS Layer* is where the simulation codes run, data are stored, etc. A failure or an error-state occurring at one layer can propagate causing additional issues.

To better understand those errors, let us consider the log files of the failed run in Appendix A.2. The first failure encountered at line 43 was caused by “bpappend” due to a missing path in the environment variables of the cluster. The missing path caused all “bpappend” and “bp2h5” operations to fail, therefore no images were generated, which resulted in the visualization job (AVS) failing to start, leading to a workflow execution failure. Note that the first failure was encountered only 1 minute after the execution has started, and yet the workflow ran for 52 minutes before it was halted due to the failures.

For the classification in table 2.1, we only list the descriptions of what we consider the root-cause of failure for each failed run, and not all the failures encountered during that run. For example, the run listed in Appendix A.2 would have the root cause classified as “missing modules/libraries/paths”, and not “Service not responding”.

Table 2.1: Root-cause analysis

Description	Faults %	Total Nb of faults encountered	System(s) on which Failure Occurred	Example
Deadlock	5%	4	All	Workflow Design Error causing deadlock
Missing Modules/Libraries/Paths	12%	11	Supercomputer Analysis Cluster	Incorrect Path on cluster (example run in Appendix A.2)
Data Movement Failure	8%	7	All	Moving data files across cluster fail or checksum incorrect
Incorrect Input	8%	7	Staging Cluster	User entered incorrect workflow parameters
Incorrect Output	9%	8	Supercomputer	File generated are corrupted
Authentication Failed	7%	6	All	Archive command failed because the HPSS authentication was unsuccessful not work
Job Submission Failed	4%	4	SuperComputer	Job not added to queue
Service not Reachable / Responding	6%	5	Analysis Cluster	AVS died
Node Crash / Down	8%	7	All	Supercomputer restarted and workflow lost connection
Network Down/Error	4%	4	All	Network outage caused the workflow to lose connection with the SuperComputer
Out of Disk Space	2%	2	Analysis Cluster	Partition was full, files could not be written
CPU time limit exceeded	10%	9	SuperComputer	Time reserved for simulation insufficient
File Not found	10%	9	SuperComputer Analysis Cluster	Data files were placed in wrong location
Other (Out of Memory, Job Stuck in Queue, Uncaught Exception…)	7%	6	All	Job priority too low and not executed.
Total	100.00%	89		

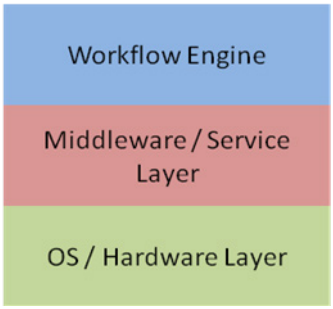

Figure 2.5: Execution Environment Layers

Run-time failures can be quite expensive. For an XGC monitoring workflow [36], the workflow runtime can vary between two and ten hours depending on the simulation with about 2/3 of the time being spent on supercomputers. We found that on the average XGC workflows fail about 9% of the time, wasting up to 7% of allocated supercomputing time. When one considers more complex situations, such as coupled workflows [39], where two different simulations exchange information, and/or one workflow controls both the simulation and the analysis, and the runtime can be as long as two days, workflow failures becomes even more of an issue.

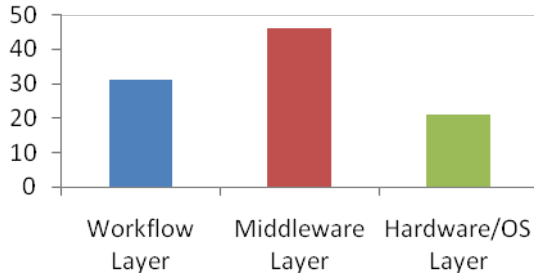

Figure 2.6: Failure percentage by Layer

Figure 2.6 provides a layer-oriented analysis of the data in table 2.1. Notice that nearly 70% of the originating issues occur either at the Hardware/OS layer or the Middleware layer. These error-states/failures can be difficult to detect. Also, in our experience, most of the time they do not propagate to the workflow layer in time, or with sufficient information attached, for the workflow to actively address the impact and resume execution. Therefore, workflow-level fault tolerance mechanisms need to be supplemented with methods that actively monitor lower layers and communicate adequate information to the workflow-layer in time for a recovery action to take place.

Now let us consider the different types of resources/clusters used for our workflow executions, and the time spent per run on each resource. For simplicity we will assume an average workflow runtime of 6 hours. The resources can be divided into 3 categories:

1. The staging cluster: A small sized cluster with 32 nodes. It is responsible for launching the workflow, preparing the scripts and input files, and submitting the job to the supercomputer. Then monitor for output files, and transfer them to the processing cluster. Around 10% of the workflow run time is spent at this stage, so considering an average running time of 6 hours, the time spent on this cluster is approximately 36 minutes.
2. The supercomputer: The majority of the workflow run time, around 70%, is spent on the supercomputer. That means approximately 252 minutes out of our 6-hour simulation runtime is spent at this stage. The supercomputer currently being used is Jaguar located at ORNL [23].

3. The analysis and visualization cluster: Approximately 20%, or 72 minutes, of the 6-hour simulation time is spent analyzing and visualizing the data. This cluster is made of 162 nodes.

2.4.4 Failure Analysis

Figure 2.7 below represents the proportionally projected average run time per computing resource based on the overall run time of production workflows. For the workflow runs examined, the run time ranges between two and ten hours. Notice that over 50% of a simulation run time is spent on the supercomputer.

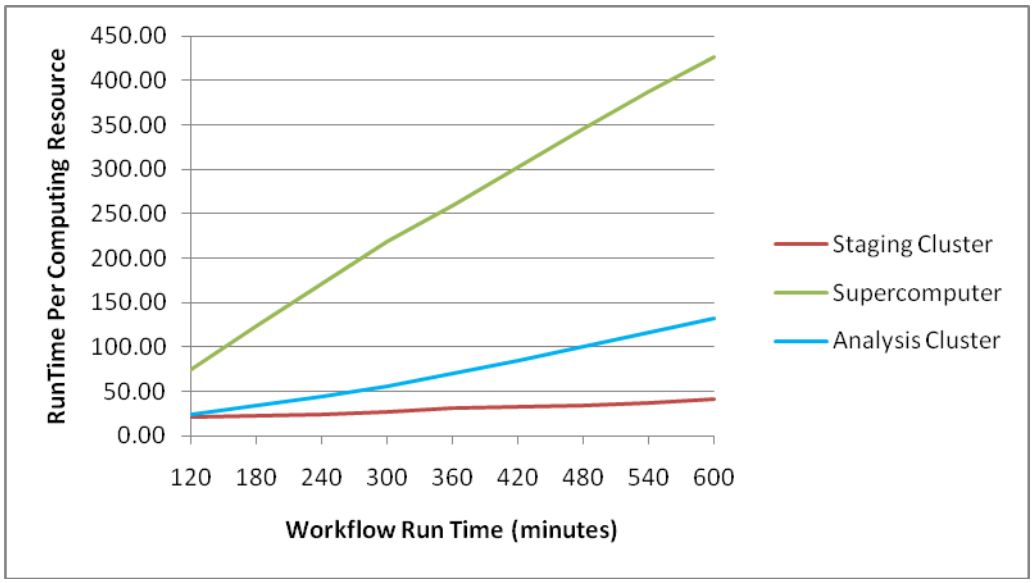

Figure 2.7: Average run time per Computing Resource based on the total run time

Figure 2.8 shows the percentages of failures and the time they occur based on the examined logs, adjusted for a 6 hour workflow runtime. The blue bars represent the failures happening

at the staging cluster, the red bars represent the failures that occurred on the supercomputer, and finally the green bars represent the errors that occurred on the analysis cluster.

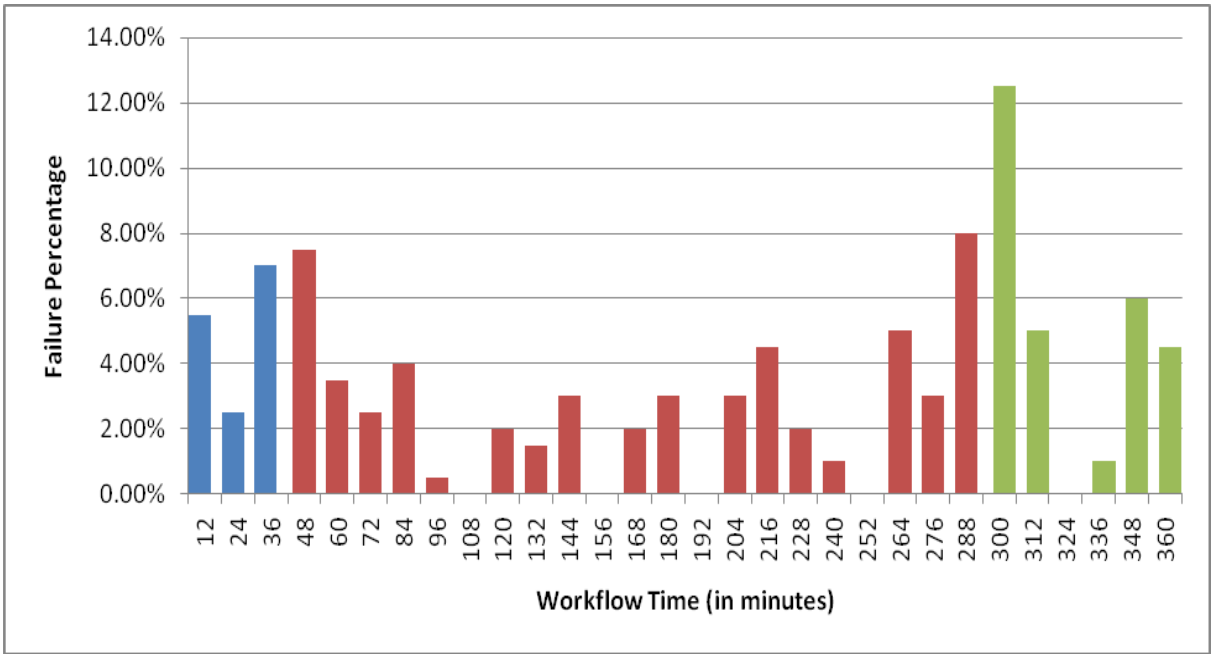

Figure 2.8: Time of failure occurrences

The failure percentages shown in figure 2.8 map onto the root causes which lead the workflow to fail either at that time, or later during the run. For example, a failure occurring at minute 228 might cause the workflow to halt then, at a later time, or it might not halt the workflow at all. In the last case, this means it will be up to the end user to detect it. For simplicity, we will consider that the time of at which failure occurs is the time when the workflow halts.

If we take as example a workflow with runtime duration of 6 hours, based on figure 2.7, the workflow would run on the staging cluster for 36 minutes, on the supercomputer for 252 minutes, and on the analysis cluster for 72 minutes. Now based on the failure rates noted in table 2.1, and the times of errors noted in figure 2.8, we can estimate that the average time

lost per workflow run (by taking the total run time of successful and failed runs and dividing by the number of successful runs) to be 1.93 minutes on the Staging Cluster, 13.54 minutes on the supercomputer and 3.87 minutes on the analysis cluster for that particular run.

Finally let us consider the average time lost per workflow run based on the duration of each run. As we mentioned earlier, the runtime ranges between two and ten hours, but as more complex workflows are introduced, the run time will go up, lasting up to several days. Figure 2.9 below shows the average time lost per run based on the duration of the run (calculated by taking the total run time of successful and failed runs and dividing by the number of successful runs).

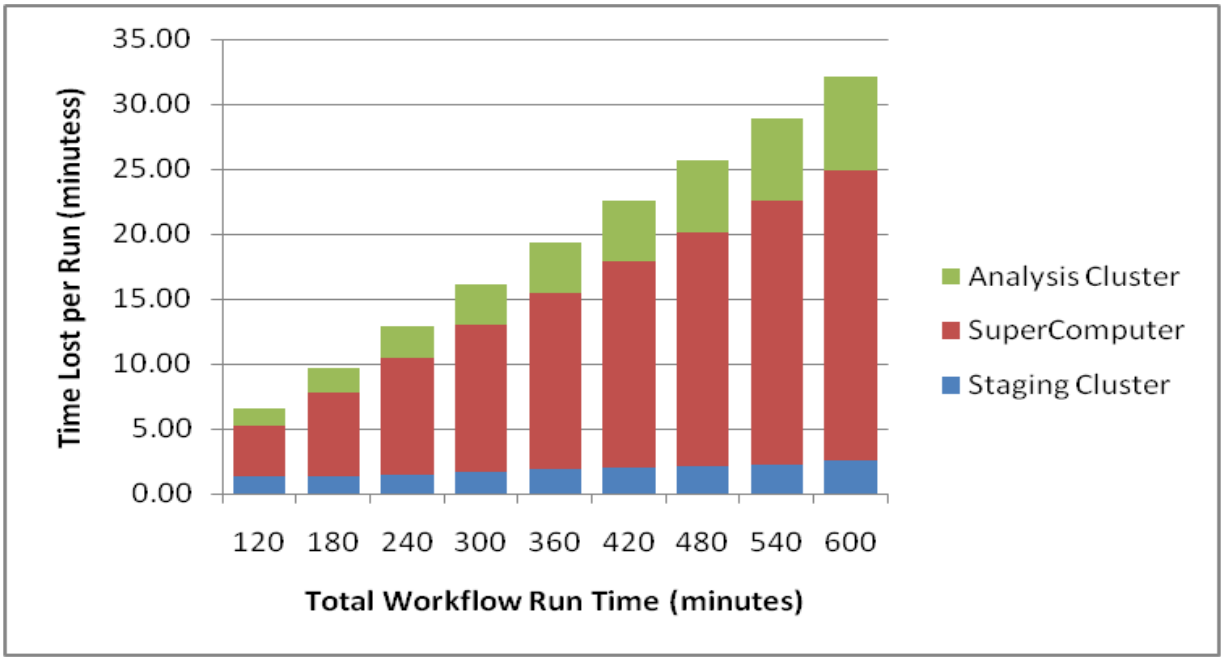

Figure 2.9: Time lost per computing resource (per run) based on the workflow run time duration

Chapter 3

Fault Tolerance mechanisms in scientific workflow management systems

3.1 Overview

This chapter presents an overview of the most commonly used scientific workflow management systems and their fault tolerance capabilities. A large percentage of these workflow management systems are grid based and generally rely on fault tolerance capabilities presented by the Grid framework. The fault tolerance capabilities of scientific workflow management systems can be divided into two main categories: 1) checkpointing based fault tolerance, and 2) Retries and/or alternative versions based fault tolerance.

3.2 Checkpointing

Checkpointing is defined as storing recovery information of a process, so that in case of a failure, the process can be restarted from the last saved state [42]. The recovery information includes the states of sub-processes, and the intermediate data and meta-data generated. Those states are called checkpoints.

The same basic principles apply to scientific workflows. In the case of Kepler, the sub-processes are actors within a workflow, and the intermediate data or meta-data is generated and consumed by each actor. Below is a list of the most commonly used workflow management systems that support checkpoint based Fault Tolerance, and their limitations.

1. **DAGMan.** DAGMan [43] is a meta-scheduler for Condor [44]. It manages dependencies between jobs at a higher level than the Condor scheduler. DAGMan bridges the scientific domain and the execution environment by automatically mapping the high-level workflow descriptions onto distributed infrastructures. Several workflow management systems use DAGMan, such as Pegasus (Planning for Execution in Grids) [45] and P-Grade [46].

   DAGMan offers a checkpointing based fault tolerance mechanism [47] which benefits all the scientific workflow management systems that use DAGMan as a scheduler. With the help of DAGMan, workflows can recover from machine crashes and network failures. They can detect authentication errors, idle staging job submission faults, job crashes, input unavailability and data movement faults. Recovery is effected with the help of DAGMan's Rescue DAG [48]. When the workflow abnormally terminates, a Rescue DAG is generated containing the tasks not yet finished. By executing the Rescue DAG instead of the original workflow, successfully completed portions are not re-executed.

2. **Swift.** Swift [49] is an open source workflow management system developed at the University of Chicago to help manage large scale science and engineering grid based

workflows. It uses Karajan [50] for execution engine and Falkon [51] for job submissions. Swift stores a “restart log” of workflow executions, allowing it to resume the computation in case of an early termination. They extend the rescue-DAG approach using cashing strategies, by storing and reusing the computation results of previous workflow components invocations.

3. **Askalon.** Askalon [52] is an open source workflow management system that provides a set of high-level services for transparent Grid access, including a sophisticated scheduler for the optimized mapping of workflows onto the Grid. Askalon provides workflow level checkpointing and migration, in which it saves the workflow state and the intermediate data at the point when the checkpoint is taken. It then allows the execution to be resumed on either the same grid resources or migrating the workflow to different grid resources.

4. **Trident.** Trident [53] is a scientific workflow management system developed at Microsoft Research. Trident provides a workflow level checkpointing mechanisms. It allows the user to restart a failed workflow from the last successful checkpoint, and gives the user the option to allocate different resources for that workflow restart, similar to the migration option available in Askalon.

5. **LEAD.** LEAD [54] is a grid based scientific workflow management system aimed for Environment and Atmospheric simulation and analysis. The authors in [33] present an

extension to LEAD in which they provide Fault Tolerance support using checkpointing and migration.

6. **Others.** Several other workflow management systems provide similar checkpointing capabilities. For example the authors in [55] present a checkpointing mechanism that incrementally records the state changes which can be undone in a rollback in case of a fault. However this solution can only be used at runtime and doesn't provide persistent state storage for a full restart after a workflow crash.

The authors of [56] present another checkpointing mechanism that saves the intermediate results at the middleware level, by extending the anthill [57] execution engine. In [58], the authors present a workflow level checkpointing mechanism in which every workflow node creates its own local checkpoint autonomously after its enactment.

3.3 Fault tolerance based on Retries and Alternative Versions

The notion of Retries/Alternative Versions [59] [60] recovers from a failure in the execution by either re-executing the failed step, or by executing an alternative version of the failed step. This also includes the execution of alternate versions in parallel with the hope that at least one version would deliver a valid result. Alternative versions can be either identical, or they can be independent versions that are functionally equivalent.

These notions have been applied to several occasions to scientific workflow management systems. Below is a list of the most common ones.

1. **Pegasus.** As we mentioned earlier, Pegasus (Planning for Execution in Grids) [45] relies on DAGMan for checkpointing capabilities, but it also offers workflow level redundancy using retries, resubmission and task migration.

2. **Taverna.** Taverna [61] features a three-layered architecture: The "application data flow" layer provides a user's perspective of a workflow, hiding the complexity of interoperation of services; the "execution flow" layer is responsible for workflow scheduling, service discovery, data, and metadata management; and the "processor invocation layer" is responsible for the invocation of concrete services. Fault tolerance support in Taverna is limited to retries and alternative versions [62] at both the services level and the workflow level. Several retry types are supported such as exponential back-off of retry times.

3. **Triana.** Triana [63] is an open source problem solving environment. It combines an intuitive visual interface with powerful data analysis tools. It is used by scientists for a range of tasks, such as signal, text and image processing. Triana includes a large library of pre-written analysis tools and the ability for users to easily integrate their own tools. Support for fault tolerance is generally user driven and interactive in Triana. For example, faults will generally cause workflow execution to halt, display a warning or dialog, and allow the user to modify the workflow before continuing execution.

Machine crashes, network errors, missing files and deadlocks are recognized by GridLab GAT [64] and the Triana engine, respectively, but recovering from or preventing them is not supported. Data movement and input availability errors are detected by the Triana engine but are not recoverable. Workflow level light-weight checkpointing and workflow restart are currently supported.

4. **P-Grade.** P-Grade [46] supports retries and resubmissions with the help of its resource and task manager (gLite [65]). It also supports light weight checkpointing at the workflow level.

5. **Askalon.** Askalon [52] supports recovery of certain types of errors such as data movement faults, unavailable input data, failed authentications and a few others. However it does not support user defined exceptions in its current version.

6. **Swift.** Among the middleware components that Swift [49] relies on, it also uses Globus services [66] which inherently provide fault tolerance capabilities to Swift such as retries and execution migration. Beyond that fault tolerance support is limited to checkpointing at the middleware level.

7. **LEAD.** The authors in [33] present a fault tolerance extension to LEAD called “Over Provisioning”, which is an alternative versions implementation for LEAD.

8. **Others.** There are several other common scientific workflow management systems, such as VisTrails [67], Escogitare [68], and View [69]. They offer similar capabilities to other WFMS, but each is adapted to a particular science niche and depends on different types of infrastructure. Vistrails is an open-source scientific workflow and provenance management system that provides support for data exploration and visualization. It is the first system that supports provenance tracking of workflow evolution in addition to tracking the data product derivation history. Escogitare is used for bioinformatics and geoinformatics experiments and uses the BPEL language [70] for describing workflows. Most of these systems provide partial fault tolerance support, either by the workflow system or the underlying infrastructure. For example, Escogitare mainly relies on the *.catch.* operation of BPEL [70] to handle certain types of errors. View presents a task level exception handling language, which defines the alternatives to be executed in case of a failure of a specific task, similar to our previous work described in [9].

3.4 Summary

Table 3.1 below lists the scientific workflow management systems mentioned above and summarizes the fault tolerance capabilities of each of those systems. Note that Kepler, in its current release (2.0), does not support any fault tolerance capabilities, and current failures are addressed on a case by case scenario by the developers. In order for Kepler to remain competitive with other scientific workflow management system, it needs to be extended to provide such capabilities. The last entry in table 3.1 represents the extended capabilities of Kepler which are introduced by the fault tolerance framework presented in this dissertation.

Table 3.1: Scientific Workflow Management Systems F/T Capabilities

	Retries		Checkpointing	Error Detection	
	At W/F Level	At Middleware		Preemptive	Backward
Pegasus	Yes	No	Yes	No	No
Taverna	Yes	Yes	No	No	Yes
Triana	Yes	Relies on Grid	No	No	Interactive
Vistrails	No	No	No	No	No
Escogitare	No	Relies on Grid	No	No	No
Swift	No	Relies on Grid	Yes	No	No
Askalon	No	Relies on Grid	Yes	No	No
P-Grade	Yes	Yes	Yes	No	Manual
PTIDES	No	No	Yes	No	No
Trident	No	n/a	Yes	No	Yes
LEAD	Yes	Yes	Yes	No	Yes
Anthill extended	Yes	n/a	Yes	No	Yes
VIEW	Yes	n/a	No	No	No
Kepler 2.0	No	No	No	No	No
Kepler with Suggested F/T Framework	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**

As mentioned earlier, most of the workflow management systems are grid based and they mostly rely on middleware grid services for fault tolerance support. Note that none provide preemptive failure detection, and there is limited backward detection, i.e., he ability to determine the failure source once an error has occurred. Our framework, discussed in Sections Five, Six and Seven, aims to provide such capabilities.

Chapter 4

Kepler provenance framework

4.1 Description

The Kepler provenance framework [4] was developed to help analyze and process the data generated by Kepler workflows. The provenance data recorded are also used by the fault tolerance framework as discussed in the following sections. Following is a brief overview of the different types of provenance and of the Kepler provenance framework [13] [14].

Data provenance refers to the origin and the history of the data and its derivatives (meta-data). It can be used to track evolution of the data, and to gain insights into the analyses performed on the data. Provenance of the **processes**, on the other hand, enables scientist to obtain precise information about how, where and when different processes, transformations and operations were applied to the data during scientific experiments, how the data was transformed, where it was stored, etc. In general, provenance can be, and is being collected about various properties of computing resources, software, middleware stack, workflows themselves, etc.

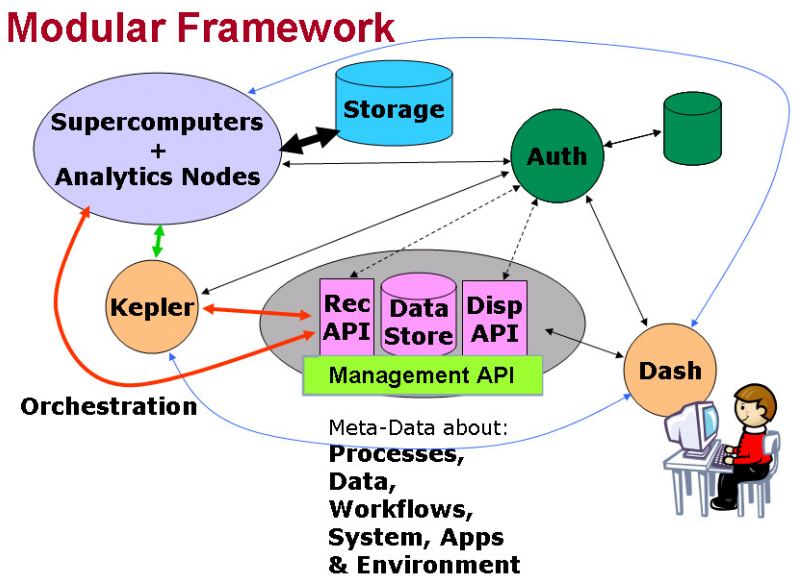

Figure 4.1: Provenance framework

The Kepler provenance framework developed by the SPA team is illustrated in figure 4.1. At the core is the provenance database with its APIs. The generality of the design allows insertion of another database (or another storage option) of choice. API has three key components: (1) Kepler, its actors, and external scripts use a **Recording API** to collect and save provenance information; (2) a **Query API** provides different query capabilities for dashboards, query actors in Kepler, and SQL statements; and (3) a M**anagement API** for Provenance Store maintenance. Currently, recording API is actually part of the Kepler.

The database stores all provenance information. The architecture has one important requirement of the database: it must be self-contained, but it can be distributed. Simple items like annotations, user identifiers, timestamps, etc., are stored in closely corresponding SQL data types, such as numeric or varchar. More complex items are stored as BLOBs. For example, the most general type of a Kepler token (item passed between actors) can be a Java

Object. In this case, if the user wishes to save all tokens produced by the actors in his workflow, he/she would need to provide serialization routines to store and retrieve the relevant Java Objects. An alternative is to use database stored pointers to reference the provenance information outside the database, such as Java Objects or workflow configuration files, pictures, movies or application source code. However, this solution may require a mechanism to reference data across machine boundaries. Self-containment guarantees that modifications to provenance information are done only using the Provenance Store Recording API, and that full set of information is preserved.

The Kepler workflow management system implements a Provenance Recorder. It is a set of listeners, or hooks – that saves token-based provenance from all (internal) workflow components into the Provenance Store. Depending on the granularity, that data may be recorded for all actors in the workflow, or some subset, e.g., only top-level composites. The **recording API** additionally supports input from components external to Kepler. These components are usually Python or shell scripts called by actors running in a workflow. Furthermore, these scripts may execute on a machine other than the one running Kepler. An example is the performance parameters that may come from software instrumentation and performance tools. In this dissertation we are particularly interested in the information stored about actor firings and tokens, which is the basis of the analysis routine mentioned later.

The **Query API** provides a read-only mechanism to retrieve provenance information from the database. It provides a range of capabilities from notifying applications as provenance is

recorded (e.g., a web based dashboard) to querying details about past executions. In addition to providing current workflow status, applications can query the Provenance Store about past executions. The Query API contains an SQL interface to support these types of queries; it can retrieve data from the database given appropriate authorization. Finally it also provides a callback mechanism for applications wishing to receive real-time provenance updates.

The **management** interface provides user administration and maintenance operations. User administration includes, adding and deleting users, modifying user passwords, modifying user access rights, and specifying the set of accessible workflows.

Security of the system has always been of importance. For example, in the current context of its use, the challenge is automated communication and exchange of data with other government labs. We are in the process of implementing a certificate-based security envelope that will allow inter-laboratory exchanges. Furthermore, we are increasingly concerned about the sharing of the provenance data. We realize that workflow meta-data and provenance information may have as much value as the raw information. Typically, sensitive information produced by a computational processes or experiments is well guarded. However, this may not necessarily be true when it comes to provenance information. The issue is how to appropriately share confidential provenance information. We developed a model for sharing provenance information when the confidentiality level is dynamically decided by the user [71].

4.2 Provenance database

Figure 4.2 shows the provenance database schema used by the Kepler provenance recorder.

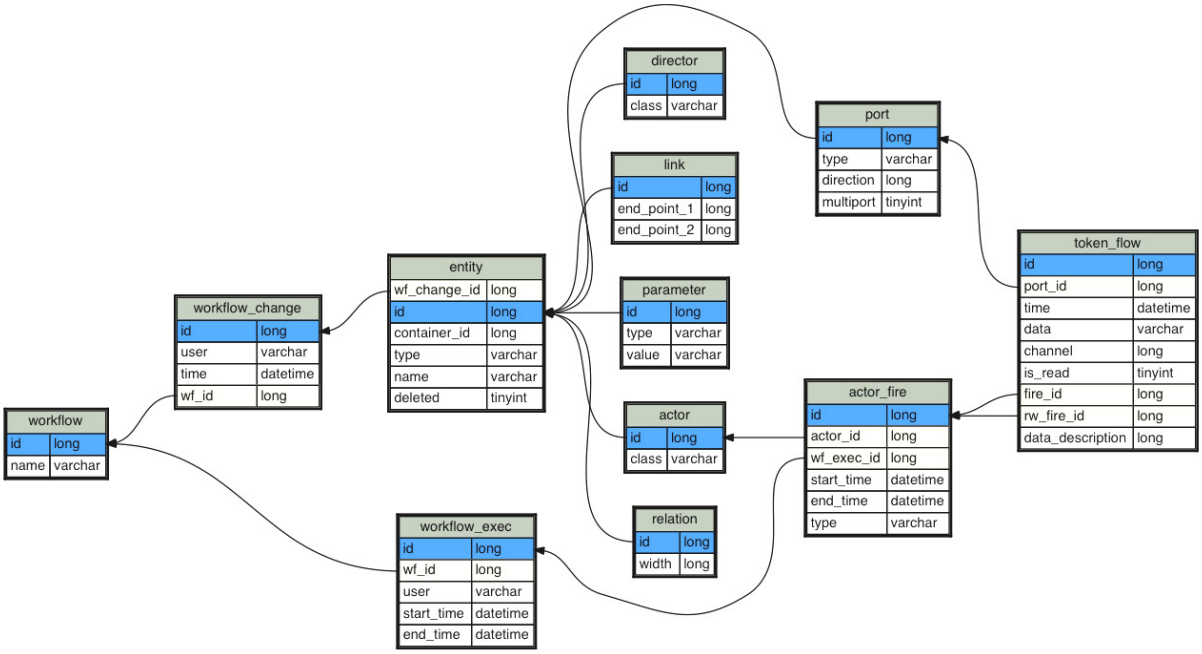

Figure 4.2: Provenance database schema

The provenance framework records the provenance data for all the entities of a workflow as shown in the database schema. This includes recording the director parameters, the actor parameters, firings, ports, tokens generated and consumed, etc. For the purpose of this dissertation, we are particularly interested in the tokens generated and consumed, since they will help us establish a lineage of the execution, and create an execution flow. This is important for the Checkpointing solution presented in chapter 6 because it will help us re-populate the actor queues based on the last successful checkpoint. This will be discussed in details in chapter 6.

4.3 Token tracing

In general the amount of data collected by the Kepler provenance framework is large and grows significantly with the size and complexity of the workflow. Usually the information of interest is only a small fraction of what's being collected. Having the scientists search through that data is time consuming and negates the purpose of data provenance meta-data, therefore the need for automated, flexible and extensible mechanisms for mining the provenance data.

For example, if a scientist wishes to further investigate or analyze an image that was generated by the workflow, he/she may need to first locate the file that contains the data behind that image. This is not an evident task since a single run can have hundreds of images associated with it. Furthermore, the relationship between images and the data files may not be one-to-one. This means that one data file may contain the data for several images. Consequently the scientist needs to both locate the data files on the disk, and to examine the contents of the data files before it can be determined which one contains the data of interest. That can generate a lot of overhead, especially for complex workflows such as those discussed in section 2.4.1 which has multiple codes integrated in it.

To illustrate and discuss development of provenance analysis algorithms and their insertion in to our Kepler-based provenance framework we will use image data files and archived data files tracking as examples [13].

The challenge is to identify and track the data files that generate images displayed on the Dashboard [5]. The solution developed checks the tokens generated and consumed, and creates a reverse graph of the token flow, starting with the image name, and ending with the data file(s). However tracking archived files was more challenging because the token trace is not linear when tracing forward through the branching trees. Instead of tracing backwards, we need to trace forward through the branching trees. Figure 4.2 below details the different steps required for successfully finding an archived data file.

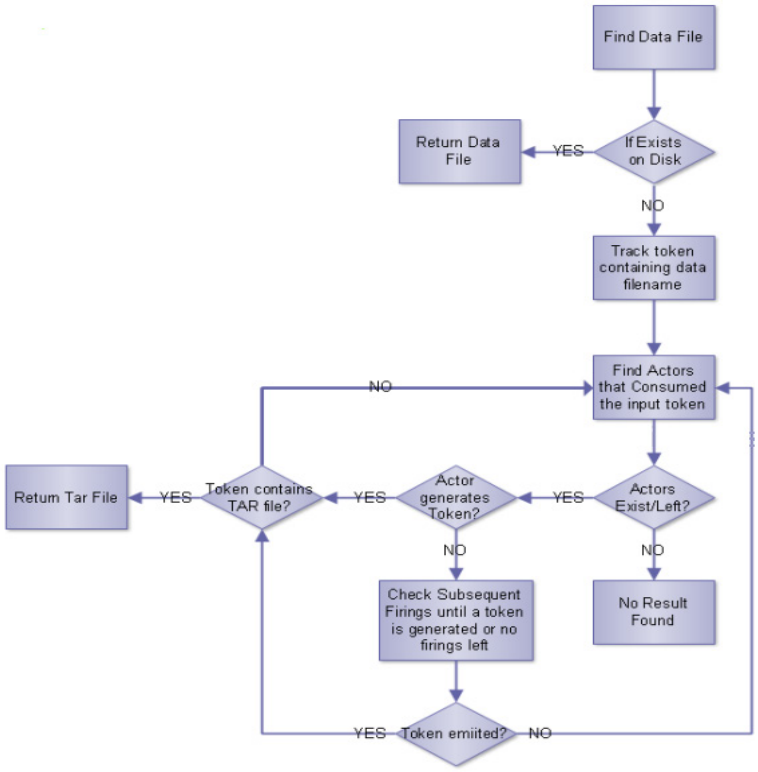

Figure 4.3: Tracking Archived Data Files [13]

The work presented in this section is being included in this dissertation because it explains how the token trace is established. This solution was developed to track the data files generated by the workflow, but the same principles apply for populating the queues for a

checkpointing based restart. The upcoming sections shall discuss in detail the role the provenance framework plays in our proposed solution.

Chapter 5

Forward Recovery

5.1 Motivation

The environment in which workflows under consideration here execute can be divided into three layers: The workflow layer, the middleware layer, and the OS/hardware layer. The workflow layer, or the control layer (figure 1.1), is the driver behind the wheel providing control, directing execution, and tracking the progression of the simulation. The framework we are proposing has three complementary mechanisms: a) a forward recovery mechanism that offers retries and alternative versions at the workflow level, b) a checkpointing mechanism, also at the workflow layer, that resumes the execution in case of a failure at the last saved good state, and c) an error-state and failure handling mechanisms to address issues that occur outside the scope of the Workflow layer. This chapter describes the first component if that framework, the Forward Recovery mechanism. Chapters six and seven describe the remaining two components.

Forward recovery mechanisms attempt to compensate for failures and keep the workflow execution going without a major externally visible interruption such as re-execution of the whole workflow. To confine failures, a system must automatically recognize error-states, for

example by checking execution results for correctness. There are two main approaches to detecting error-states caused during execution faults:

(i) Acceptance testing of the results via executable assertions [59]. An acceptance test may determine functional equivalence, or may determine correctness measured against an always-correct result, and

(ii) Use of alternate version(s) [59] [60]. Alternate versions execute in parallel and each delivers a result. Alternative versions can be either identical, or they can be independent versions that are functionally equivalent.

Results delivered then need to be compared to determine which one is correct, and then either the correct result is passed on to the rest of the workflow, or a "graceful" failure exit needs to be taken. The best-documented techniques and most commonly used for tolerating faults in software-based systems are the Recovery Block (RB) approach [60] and N-version programming (NVP) [72] In either case, there needs to be a decision mechanism which deals with the situation where it may not be possible to recover from a failure. In this case one also has to worry whether failures that might occur are correlated or not. Correlated failures can reduce effectiveness of the failure or error-state detection mechanisms [59].

5.2 Contributions

The main contribution of this part of work is a Retry/Alternative Version mechanism for the Kepler workflow management system. Although principles have been known for some time and have been extensively studied in the literature, the contribution here lies in adapting these principles to modern day workflow systems, and extending them to interact and

complement the other fault tolerance mechanism discussed in the next two chapters, in order to present an entire fault tolerance framework that would address most if not all of the errors encountered in our production workflows.

5.3 Architecture

5.3.1 Overview

To implement the forward recovery part of the whole strategy, we develop a **Contingency Actor**. Its early predecessor – fault-tolerant web-services actor is discussed in [8] along with performance gains that can be achieved assuming failure independence. Several sub-workflows may be associated with this actor: one sub-workflow represents the primary set of tasks to execute. When a failure occurs, this sub-workflow can be re-executed with the original inputs if we believe that the failure is a transient one, or an alternative sub-workflow can be executed instead. Furthermore, the actor may pause the execution between sub-workflow executions if some other action needs to be taken. This work was implemented jointly by the SPA team at North Carolina State University and San Diego supercomputing center (SDSC) and was published in [12].

5.3.2 Workflow interface

Figure 5.1 shows an example configuration of the contingency actor. The primary task, implemented in a sub-workflow, called “primary-ssh”, is to *ssh* to a server and execute a script. This sub-workflow is shown in Figure 5.2 with the alternative sub-workflows accessible via tabs. If the server is down, the contingency actor will re-execute the primary-

ssh sub-workflow three times, waiting five minutes between each attempt. If the connection cannot be made after the final attempt, a secondary sub-workflow, “secondary-ssh,” is run, which *ssh-e*s to a different server. If this fails, the contingency actor runs the third sub-workflow, “email”, which notifies the user and the workflow fails.

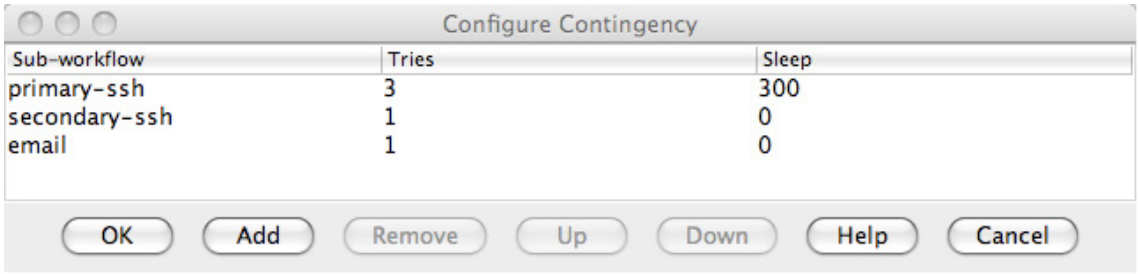

Figure 5.1: Example contingency configuration

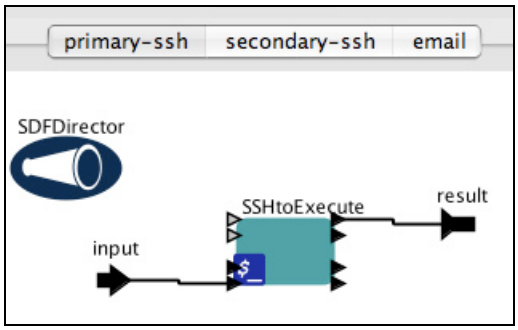

Figure 5.2: The primary-ssh sub-workflow in the contingency actor

When the contingency actor receives new input from upstream actor(s) for the first time, it executes the primary sub-workflow. The user chooses the first (primary) sub-workflow to execute: it may either be the same sub-workflow each time, or the last successfully executed sub-workflow. In the previous example, ”primary-ssh” should always be tried first since the script runs faster on that server. However, if this server is often unreachable for long periods of time, then it may be faster to skip retrying the primary server and immediately go to the

secondary server. In this case, contingency can be configured such that if “primary-ssh” fails but “secondary-ssh” succeeds, subsequent input will execute “secondary-ssh.” Furthermore the knowledge of which sub-workflow to execute first may be shared among multiple instances of the contingency actor within a single workflow.

5.3.3 Monitoring layer plug-in

To better help the contingency actor decide which alternative to choose, by determining which one has the highest rate of success, we provide an interface over which it can communicate with other components of the presented fault tolerance framework, namely the monitoring layer and the provenance database. The monitoring layer and the interaction between it and the contingency actor are discussed in detail in chapter 7.

5.4 Error-state handling

One of the classical failures is resource timeouts. Below is a simple example of how the contingency actor can address timeouts by retries or, if all alternatives fail, by pausing the workflow and notifying the user. The sequence diagram in Figure 5.3 below shows how the contingency actor can be configured to automatically increase the timeout and re-execute the script.

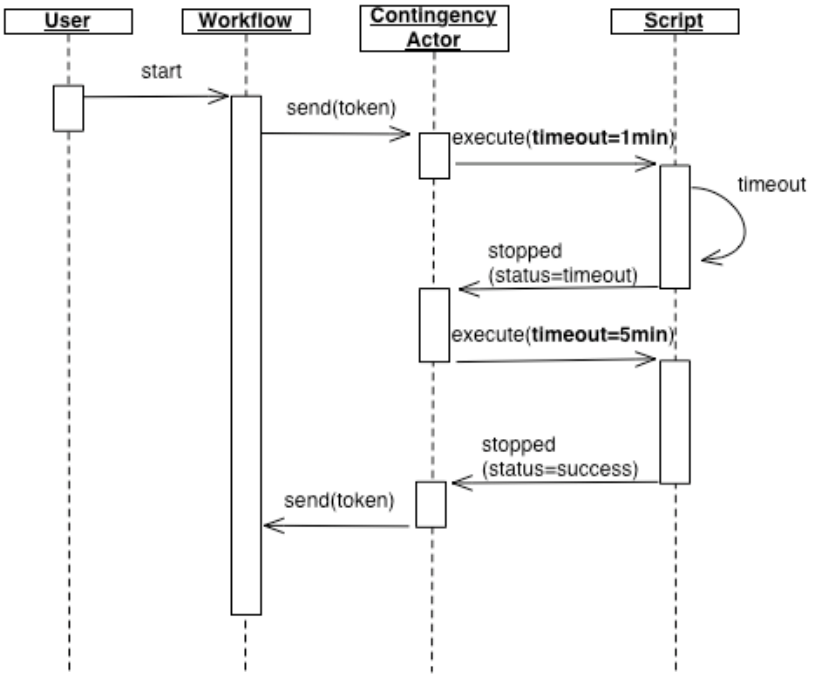

Figure 5.3: Automatic retry with increased timeout

A user starts the workflow which at some point invokes the contingency actor. The primary task of contingency executes the script with a timeout of one minute. When this timeout is exceeded, the script aborts and notifies the contingency actor. The contingency actor's secondary task then increases the timeout to five minutes and re-executes the script. The timeout is not exceeded, and the script successfully completes. The workflow then executes the remaining actors and eventually stops.

Figure 5.4 shows another scenario where the timeout parameter may be specified in the external application and not available to the workflow layer. When a timeout occurs in these situations, our architecture can notify the user, gracefully stop workflow execution, and provide smart resume once the timeout has been increased.

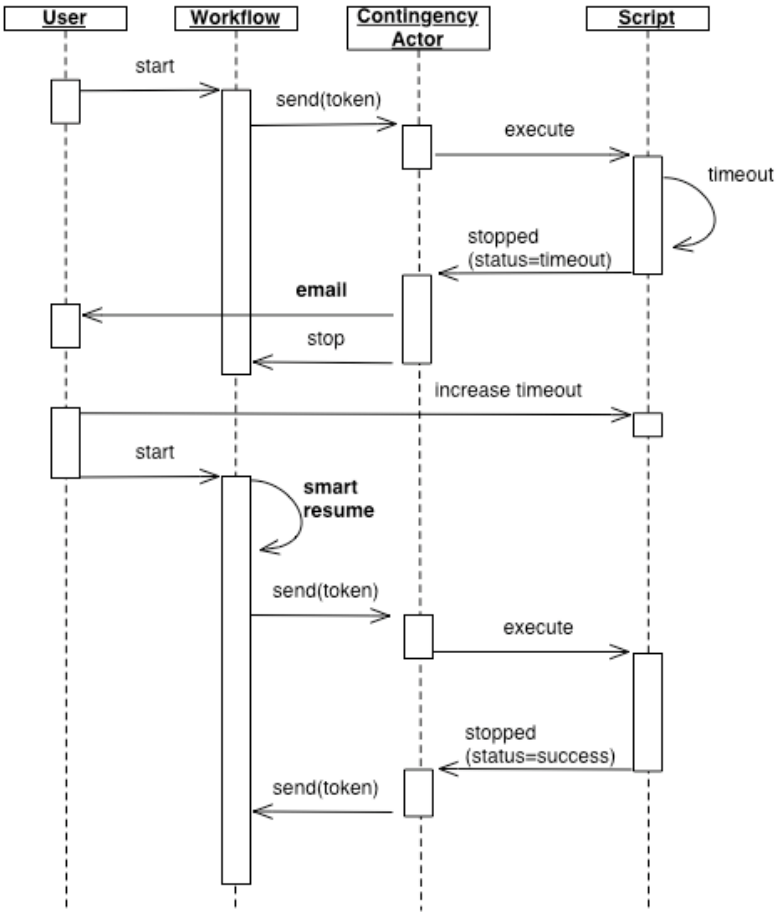

Figure 5.4: Notify user and smart resume [12]

As in the previous solution, a user starts the workflow, which eventually executes the contingency actor. In this solution, the primary task executes the script without a timeout value (specified in the workflow layer). When a timeout occurs, contingency's secondary task sends email to the user describing the problem, and stops the workflow. Once the user has increased the timeout in the script, she re-executes the workflow. During initialization, the workflow performs smart resume so that completed tasks are not needlessly re-executed.

The contingency actor's primary task executes the script, which completes successfully. The workflow then runs to completion.

Yet, another scenario is when a failure occurs executing a task which would cause subsequent tasks to fail, thus those subsequent tasks should be not executed since they may crash the workflow. Figure 5.5 explain how this scenario is addressed using the contingency actor.

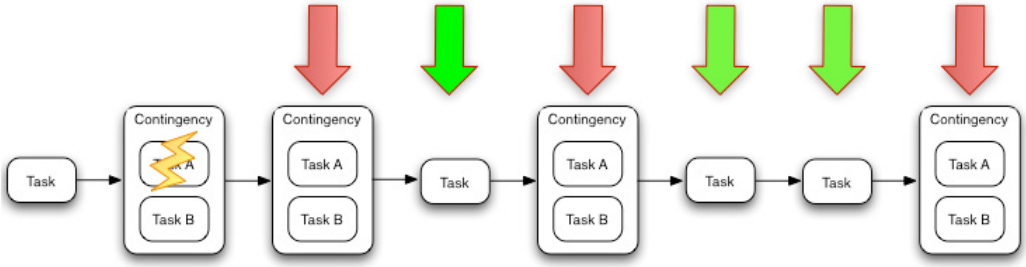

Figure 5.5: Embedded Tasks in Contingency Actors [11]

The tasks that always execute (green arrows) are unmodified, but tasks that should not be executed if an error-state occurs (red arrows) have been embedded in contingency actors. Each contingency instance contains two sub-workflows: Task A executes if there are no error-states, and task B executes when an error-state occurs. For this use case, task B is an empty sub-workflow that simply passes input data to the next task. If an error-state occurs while executing task A in the first contingency actor, task B executes. In subsequent contingency actors, task B executes instead of task A, preventing the workflow from crashing.

Chapter 6

Checkpointing

6.1 Motivation

Another approach for providing fault tolerance through backward recovery is checkpointing. Checkpointing principle is a widely used and consists of storing a snapshot of the current application state, and using it for restarting the execution in the case of a failure [73]. There are many ways for achieving application checkpointing. Depending on the specific implementation, a checkpointing tool can be classified by:

a) ***Amount of state saved*:** This refers to the abstraction level used by the technique to analyze an application. It can range from seeing each application as a black box, hence storing all application data and state information, to selecting specific relevant core data in order to achieve a more efficient and portable operation.

b) ***Automatization level*:** Depending on the effort needed to achieve fault tolerance through the use of a specific checkpointing solution.

c) ***Portability*:** Whether or not the saved state can be used on different machines to restart the application.

d) ***System architecture*:** How is the checkpointing technique implemented: inside a library, by the compiler, or at operating system level.

The motivation behind using checkpointing is to reduce the time required for a re-execution of a failed workflow. If a workflow uses checkpointing, tasks that were successfully completed before the workflow abnormally stopped may not have to be re-executed.

The solution presented allows workflow execution to be resumed, using the provenance data automatically captured by the Kepler Provenance Framework, namely the data collected in the “token_flow”, “entity”, “actor” and “actor_fire” tables (figure 4.2). Thus reducing the time required for re-execution of failed workflows.

6.2 Contributions

The challenges in extending the Kepler Workflow Management System reside in addressing two main issues:

a) Stateful actors (more precisely a combination of stateful and stateless actors) and
b) Workflow loops.

The main contribution of this work, beside the time/resources savings for the re-execution of failed workflows, is presenting a checkpointing system that (1) supports stateful multi-invocation actors, and (2) memory actor-actor data transport and communication, i.e. the data is maintained within the workflow engine and without disk persistence. To achieve that, we introduce an extension to the Kepler provenance framework which enables it of recording the internal states of stateful actors, thus allowing those states to be properly checkpointed.

6.3 Architecture

6.3.1 Extensions to the provenance framework and workflow scheduler

The checkpointing solution presented in this chapter is being developed in collaboration with other members of the SPA team from ORNL and the University of California at Davis. It extends the provenance framework presented in chapter 4 by adding details about the states of actors and events that occur during an actors' invocation cycle. Those details include adding the following relations:

- State relation. This relation stores information on whether an actor is stateless or stateful. It contains two parameters as follows:
    - Invocation ID: Tracks the ID of each invocation of an actor.
    - State: Stateless or stateful.
- Invocation relation. This relation helps the checkpointing framework determine the internal execution state of an actor. This relation contains four parameters as follows:
    - Actor Name
    - Invocation ID: Tracks the ID of each invocation of an actor.
    - State: records state of the actor, which can be *running* if the actor is currently being executed, *iterating* if the actor will run again, or *done* if all invocations have been executed.
- Event relation, to store the read and write events of actors. This relations contains the following parameter:
    - Event Type: read or write.
    - Token Value: represents the value of the token generated or consumed.

- Actor Invocation ID: Tracks the ID of each invocation of an actor.
- Order of events: used to establish an order for events during the same actor invocation.

Alongside extending the provenance framework, the scheduler was updated to initiate the checkpointing process after each round of invocations. That process traverses the current level of workflow hierarchy and captures the states of each Stateful actor. Note that checkpoints are not necessarily stored after each invocation; but only after the last invocation in a schedule. The scheduler was also updated to allow the execution to start at a specific point. This is done by traversing the schedule in normal mode, but skipping all the invocations until the failed invocation is reached. At this point, the normal execution is resumed.

6.3.2 Supported execution models

Kepler supports a multitude of execution models. The following checkpointing implementation supports two of the most common models, PN [74] and SDF [75]. Those models are described below:

Process Networks with Firings (PN): A Process Network is a general model of computation for distributed systems. Actors communicate with each other through unidirectional FIFO channels of unlimited size. Workflows can be described with the four core relations: Sub-workflow, Actor, Port, and Link. The PN execution semantics allow a high level of parallelism, i.e., all actors can be invoked at the same time. After an invocation

ended, the actor will be invoked again. This procedure stops either when the actor explicitly requests to be stopped or by reaching the end of the workflow execution. A PN workflow ends when all remaining running invocations are deadlocked on reading from an input port.

Synchronous Data Flow (SDF): Besides the data captured by the four core relations, in SDF, output ports are annotated with a fixed token production rate and input ports have a fixed token consumption rate. Both rates are associated with ports using the predicate token transfer(A,P,N). During an invocation, each actor A is required to consume/produce N tokens from the input/output port P. Another extension is the firing count of an actor that specifies the maximum number of actor invocations during one workflow execution. The predicate firing count(A,N) provides this number (N) for an actor A. Unlike in PN, where the actors synchronize themselves through channels, the execution of SDF is based on a static schedule that is repeatedly executed in rounds. The number of firings of each actor per round is determined by solving balance equations based on token production and consumption rates.

6.3.3 Checkpointing based recovery

The checkpointing based recovery can be summarized as follows. The first step is to determine the point of failure, which can be achieved by checking the provenance database for the actor that consumed a token(s) but did not generate any token(s). The next step to check the status of all other actors and classifying them into three categories: i) not started, ii) running and iii) done. For the actor(s) that haven't been started, simply mark them to run, for the actors that are running, the latest invocation is considered faulty and their effect is

undone, then they are also marked to run. For the actor(s) marked as done, there is no need to re-run them, we simply replay their output tokens.

Next we need to determine whether actors are stateless or stateful. For stateful actors, we need to restore their internal state. That is achieved by reading their internal state at the last successful invocation from the provenance database. If an actor is stateless, no further actions are needed.

Next the tokens need to be restored and the queues re-initialized with the restore sequence. Once that is done, the workflow is ready to be resumed. The checkpointing framework notifies the director that the workflow is ready to be resumed, and the director resumes the workflow. This process is summarized in figure 6.1.

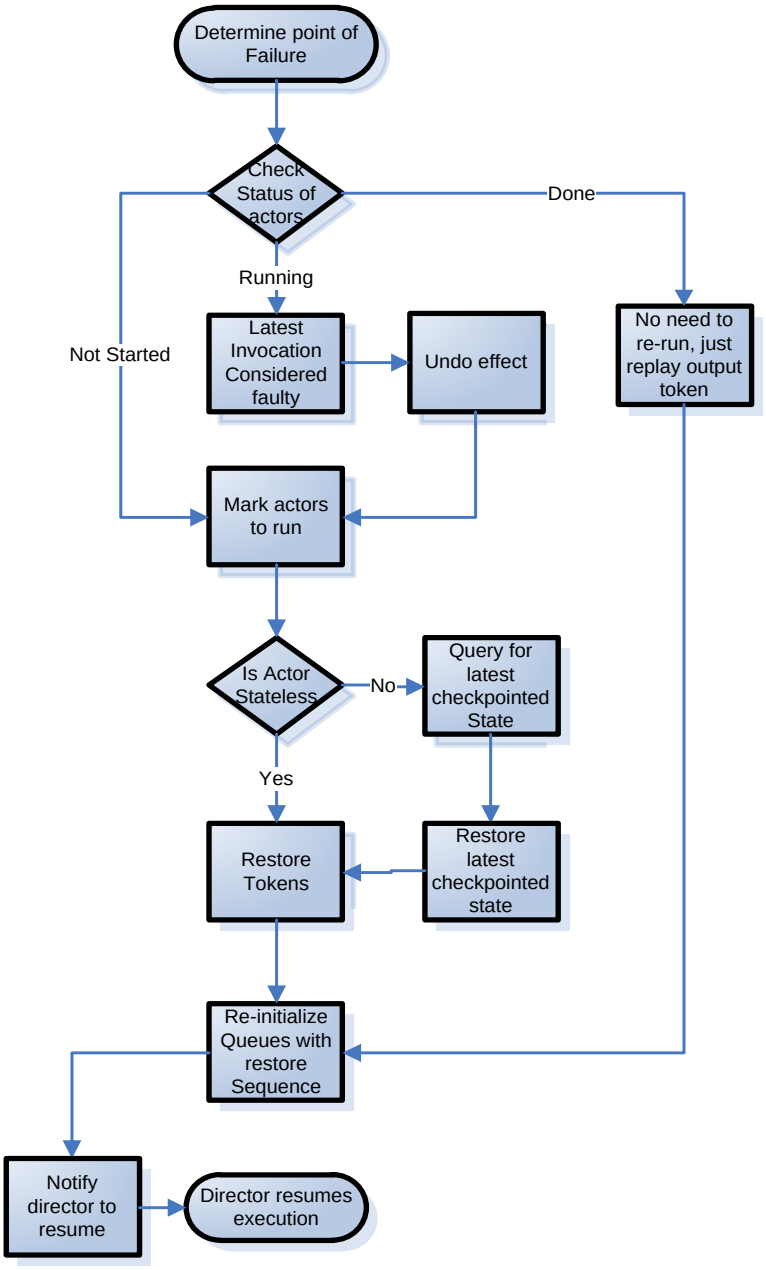

Figure 6.1: Flowchart of the recovery steps

6.4 Recovery scenario

Figure 6.2 below shows a sample SDF workflow with a mix of stateless and stateful actors.

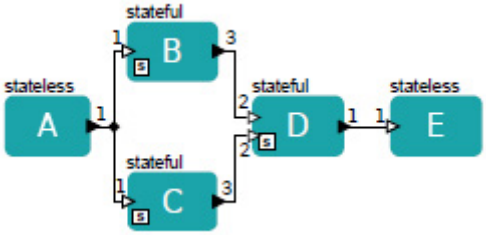

Figure 6.2: Example workflow with stateful and stateless actors

Actor A is a source actor. It will output one token per invocation and has a firing count of 2, which means that it will be invoked twice. Actors B and C consume one token each and produce 3 tokens in order. Actor D consumes two tokens per invocation from actors B and C and produces one token. Actor E just consumes one token from D.

Since this is an SDF workflow, the actors above are invoked according to a pre-computed schedule. The execution schedule is illustrated in Figure 6.3 below, with an example workflow crash at the 2nd invocation of Actor B.

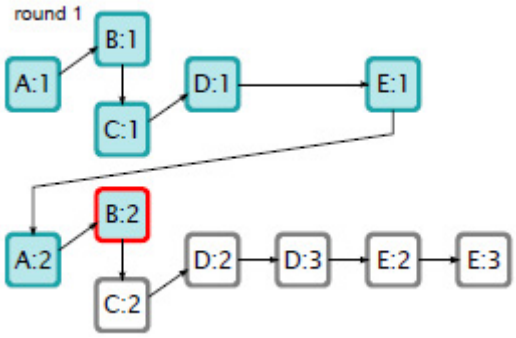

Figure 6.3: Execution order of sample SDF workflow with error-state occurring at the 2nd invocation of actor B

All invocations up to the 2nd invocation of A were successful. The 2nd invocation B was still running and all other invocations didn't run yet. Figure 6.4 below shows the failed workflow execution.

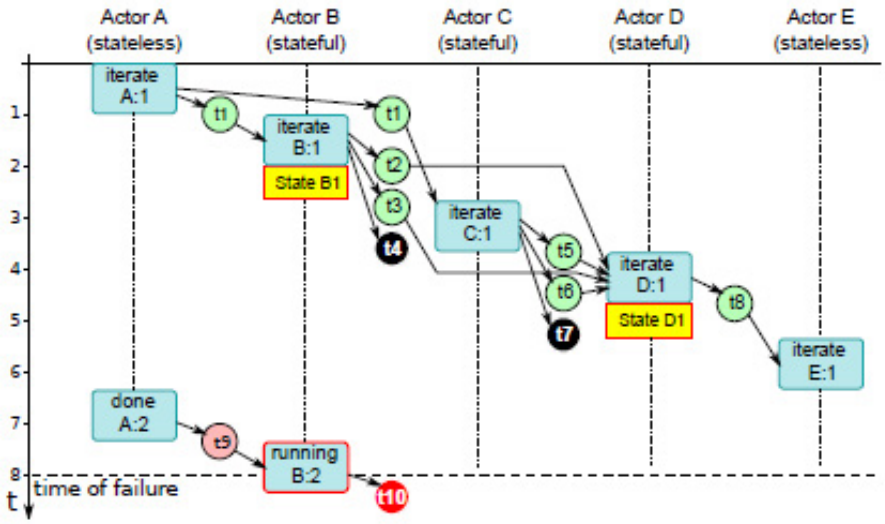

Figure 6.4: Workflow Execution up to the 2nd invocation of actor B

The recovery process can be summarizes in the following steps:

Step1 – Determine failure point: Determining the point of failure can be done by examining the actors that fired without producing any tokens, which is B:2 in this case.

Step2 – Check execution state of actors: Determine which actors are yet to run, running, or done.

Step3 - Handing stateless actors: Stateless actors are restored to their pre-failure state immediately after initialization. Actor A is considered done and will be skipped. Actor E is in a proper state after a simple initialization.

Step4 - Handling stateful actors: We check for checkpoints in the provenance database for Stateful actors. Actors B and D are stateful and checkpoints are found for them, therefore they are re-instantiated and restored to the latest recorded state. The state is shown in figure 6.4 above in the yellow box. Actor C has no checkpoints stored, therefore it needs to have its' invocations replayed.

Step5 – Restore queues: We restore all the tokens that either were for failed invocations, for invocations that were not checkpointed, or tokens that were generated but not consumed. In our example above, token t1 is restored back to the queue because the last checkpoint occurred before the invocation of C:1, token t9 is restored because B:2 was still running when the failure occurred. Tokens t4 and t7 are restored because they were generated but never consumed.

Step6 – Reinitialize director: Once all the queues have been restored. The SDF director is initialized so that execution would begin with the actor that was interrupted by the failure.

Step7 – Resume execution: Once the director has been initialized, the workflow engine is notified and the workflow execution is resumed.

Figure 6.5 represents the workflow after it has been recovered and before it is re-executed. As one can notice, the process is much simpler when compared a full re-run.

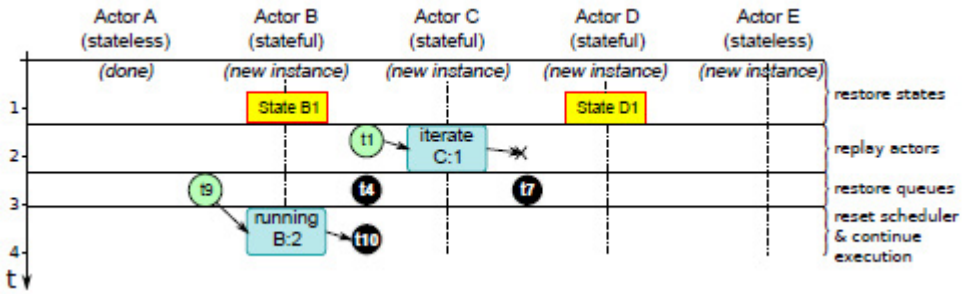

Figure 6.5: Workflow execution after recovery using checkpointing

6.5 Limitations

Current support is only for 2 models of computations, SDF and PN. In order for this framework to be utilized at a production level, it needs to support more complex models of computations, such as Dynamic Dataflow (DDF) [76]. We also assume that all faults are independent and there is no correlation among error-states, since at the current stage correlation is difficult to detect.

Chapter 7

Error-state Handling Layer

7.1 Motivation

As we have discussed in section 2.4.4, over 70% of the errors encountered in our production runs of the XGC and S3D workflows occur at the middleware and OS/hardware layer. Those errors are at best seen as failed steps in the process without additional detail. By the time the failure manifests, the outcome of that failure would either crash the workflow or produce incorrect results. Therefore a mechanism to track error-states and properly relay that information to the workflow layer is needed in order to better address and recover from those error-states. To achieve that, we introduce a variation of a watchdog process and extend it to fit our needs. We call the proposed solution the *Error-state Handling Layer.*

To recap, a watchdog process is a common tool used in computer hardware. It consists of a hardware timing device that triggers a system reset if the main program, due to some fault condition, neglects to regularly service the watchdog (writing a "service pulse" to it) [77]. The intention is to bring the system back from the unresponsive state into normal operation.

7.2 Contributions

The main contribution of this part of the work, especially when paired with the forward recovery mechanism, is to provide a pre-emptive error-state detection and recovery mechanism based on the active monitoring of external workflow components and providing results to the workflow layer, particularly the contingency actors. As we already saw in table 3.1, none of the surveyed scientific workflow management systems provide similar capabilities.

7.3 Architecture details

7.3.1 Overview

The *Error-state Handling Layer* is a module that runs separately from the normal execution environment. Its primary task is to monitor the components and processes on which the workflow layer depends but cannot monitor. For example, visualization applications that reside in the middleware layer are tracked to check their availability and serviceability. If a problem is detected, the error-state handling layer analyzes it based on its error taxonomy, and data from previous runs collected by the provenance framework, and sends an appropriate signal and possible course of action to the workflow layer to handle it.

7.3.2 Monitoring framework

The error-state handling layer is made of several interacting components. It is designed to allow layers that are to be monitored to simply “plug” in without the need to modify or extend the error-state handling layer itself. Figure 7.1 below shows the different components

of the error-state handling layer along with its interaction with the other layers of the workflow systems.

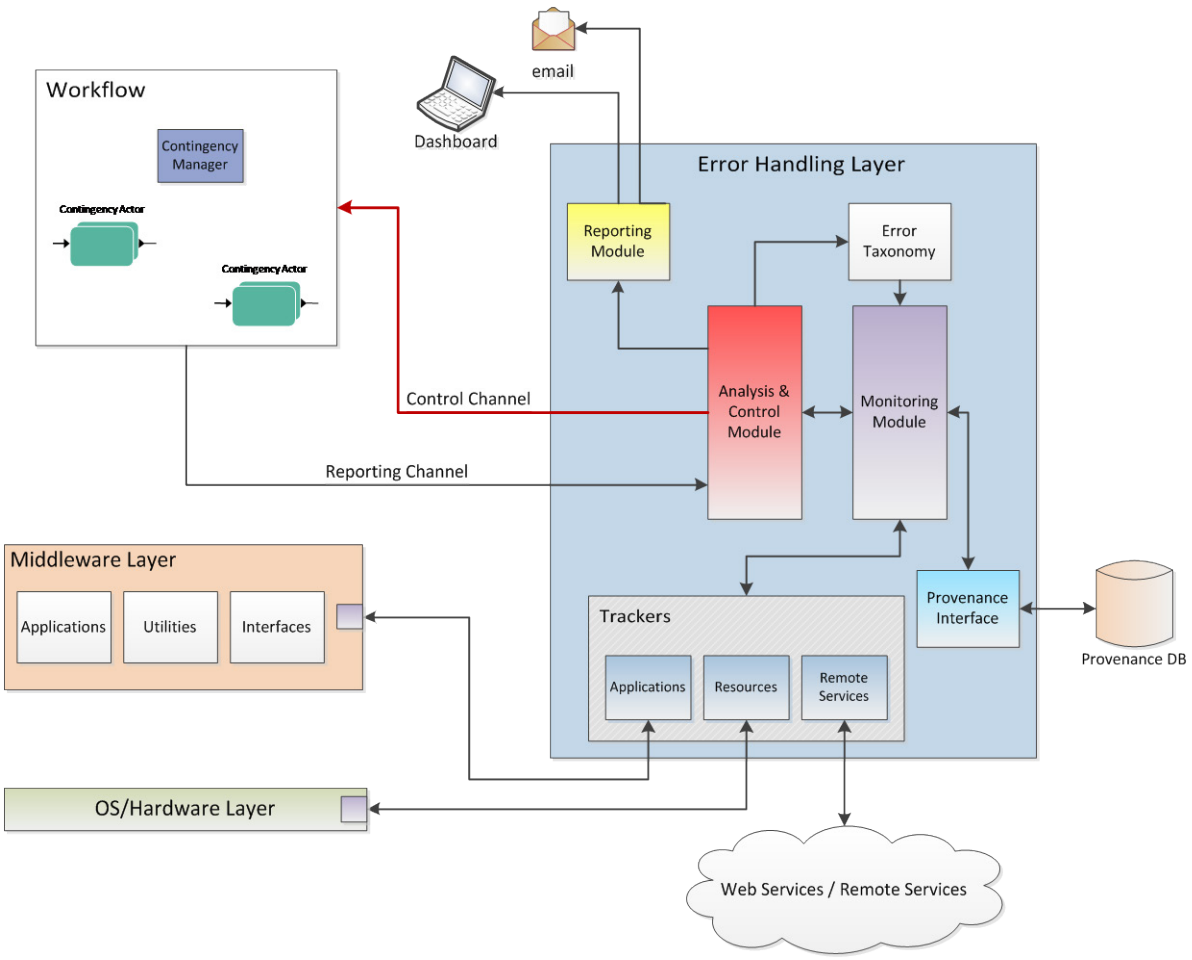

Figure 7.1: Error-state handling layer

The following is a brief description of the different components and how they interact.

(i) ***Contingency Manager:*** this module, which resides at the workflow layer, is responsible of the following:

- Upon launching the workflow, this module contacts all the contingency actors in the workflows and collects all their contingencies and parameters.
- It then establishes a socket connection to the "Analysis and Control module" in the error-state handling layer, and transmits the data collected over that socket connection.
- While the workflow is in progress, this module waits for notifications from the error-state handling layer on errors occurrence, and how to address them.

(ii) ***Analysis & Control module:*** This module, which resides in the error-state handling layer, is the main driver of that layer. It is responsible for the following:

- Collection of the contingency information from the contingency manager.
- Processing the contingency information and passing it to the error taxonomy for classification.
- Invoking the monitoring module to start monitoring the newly added components to the error taxonomy.
- Forwarding the errors and warnings found to the Reporting module.
- Instruct the contingency manager on actions to be taken once an error is detected.
- Update the error taxonomy once errors are detected.

(iii) ***Monitoring module:*** This module is responsible for monitoring the different components of a particular workflow which are listed in the error Taxonomy. Its tasks include:

- Once instantiated by the Analysis and Control module, this module will read all the components that require monitoring from the error taxonomy.

- Based on the components required, it instantiates the appropriate trackers and provides each tracker with a list of the components to monitor.
- It also checks the provenance database for data from previous runs relating to the components that are to be monitored, in order to evaluate their past performance and failure ratio.
- Based on the provenance data collected and the information returned by the trackers, notify the Analysis and Control module on the proper action to take when an error or warning occurs.

(iv) ***Provenance Interface:*** This module reads information from the provenance database relating to previous runs, failures and error-states. This information is then relayed to the Monitoring module and is used to determine the best way to handle an error.

(v) ***Trackers module:*** This module contains an extensible list of sub-modules aimed to track different layers/components that are used the workflow. For example, we would have a tracker for middleware services, a tracker for the OS and a tracker for remote/web services. Trackers can be added as needed to address different workflow requirements.

For certain tracker modules, another module is required on the target layer for better efficiency and scalability. For example, at the OS layer, it would be a lot more efficient to have a module on the target machine checking for resources, and reporting back to the error-state handling layer using a single socket connection, than having a request over the network every time a resource check is requested.

(vi) ***Error Taxonomy module:*** This module contains XML descriptions of the warnings and error-states that occur in the different layers. It is used to trigger recognition of problems. It also details the possible ways to recover from each error.

(vii) ***Reporting module:*** This module reports the failures, interrupts, error-states and warnings detected to the end user, to help him or her better track problems that might be occurring during a run. The notifications can be either posted on the workflow monitoring dashboard, or sent via emails, or via SNMP traps.

By having this modularized architecture, and having the error-state handling layer separate from the normal execution layers, we can handle situations where the entire error-state handling layer fails. When this happens, it should not interfere with the normal workflow execution. Additionally, if the error-state handling layer comes back online during the same run, it can reconnect to the different modules, read the information recorded for that run in the database, and get up-to-speed and resume monitoring.

One possible extension for the architecture presented above is adding control flows from the error-state handling layer to all other layers instead of just the workflow layer. This allows us, for example, to restart modules and components in other layers that might have become unresponsive and otherwise would cause the workflow to halt since the workflow layer can only invoke these modules.

7.3.3 Provenance based analysis

The provenance interface mentioned above enables the error-state handling layer to analyze the data collected of previous runs which would enable the “monitoring module” to make a more informed decision on the success rate of tracked components.

This is achieved by checking for runs of the workflows with the same workflow ID. Then for each contingency actor in each run, check the invocation rate and success ratio of each contingency, and finally calculate the average success ratio of each contingency and pass that value to the “monitoring module”.

7.3.4 Interaction with the Forward Recovery mechanism

Whenever a workflow is launched, two socket connections are established with the error-state handling layer. Each one serves a distinct purpose:

1. The first connection, named “Reporting channel”, is used for passing workflow meta-data to the error-state handling layer. That meta-data include the workflow ID, workflow runID, contingency actors IDs, names, number of contingency, and details about each contingency.
2. The second connection, named “Control channel”, is used to manage and direct the workflow execution progress based on the health of the monitored components. The error-state handling layer can instruct contingency actors to use the contingencies with a higher success rate, or to increase timeouts, or to pause the workflow execution if no contingencies are viable.

Note that this is not a one-to-one relationship, and the “Error-state Handling Layer” can handle multiple workflow runs simultaneously.

7.4 Recovery scenario

A common error during our workflow runs is having a middleware service become unresponsive. Without the error-state handling layer, this would lead to a timeout at the workflow level and thus causing the workflow execution to fail.

However, since the error-state handling layer would be monitoring that service, once it detects that a service is unresponsive, it would do one of the following:

- Restart the unresponsive service if it has the permissions
- Contact the contingency module in the workflow layer, which in turn would direct the contingency actor responsible for invoking the unresponsive service to use either use an alternative, increase the timeout value, or if neither options are available, instruct the director to pause the workflow execution.

Chapter 8

Framework Evaluation

8.1 Assessment environment

The framework proposed as part of this work needs to be evaluated for viability, value, validity and performance. We do that using two approaches. One is analytical – through analysis of the architecture, assumptions and use of analytical equations that capture expected behavior. The other is through a proof-of-concept (POC) framework implementation and workflow emulation and failure-injection [79]. The results from the two approaches are then compared and the overall assessment is based on the integration of the findings.

To evaluate the proposed "fault tolerance framework" through emulation and simulation, we created a hardware setup that "emulates" real operation environment, and test scenarios and workflows that resemble the setup and the workflows used in the production runs. Our POC implementation includes all three fault tolerance mechanisms discussed in chapters five, six and seven, and aims to assess the **reliability, runtime** improvements, and **overhead** of the presented solution as a whole. Failure-injection is used to simulate failures and assess

whether failure recovery mechanisms operate as expected. The hardware setup created for our tests is shown in figure 8.1.

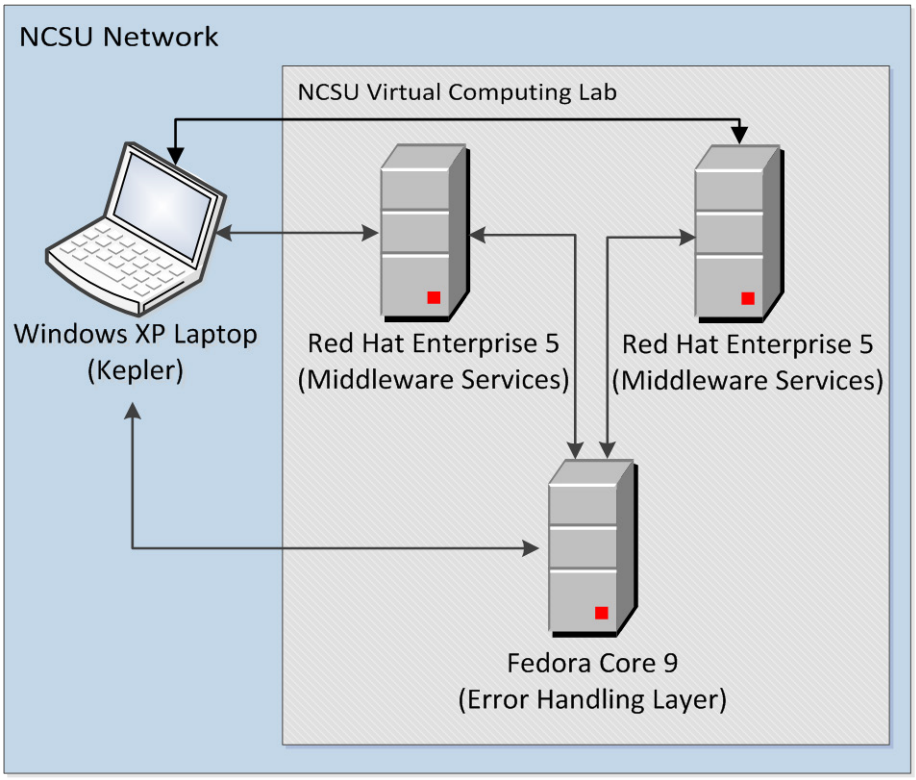

Figure 8.1: Test scenario hardware setup

We took advantage of the facilities provided by NCSU's Virtual Computing laboratory (VCL) [78] in creating this test bed. The machines setup is as follows:

1. A laptop computer running Windows XP. This machine was used to run Kepler and launch the workflows. This represents the workflow control layer (figure 1.1). This layer can be loosely coupled to the underlying resources.
2. A VCL image running Fedora Core 9.0. This was used to host the following:
    a. "Fault Tolerance framework" and all its components,

b. A MySQL database which contains the provenance database and the error and fault/failure taxonomy
c. An Apache Web Server servicing the dashboard.

3. Two (2) VCL Images running Red Hat Enterprise Linux 5 (RHEL5) representing redundant but functionally equivalent computational and analytics services. Both machines ran the following nominal services required by the workflow:
    a. Data File Manipulators: Converters (bp2h5), splitters and joiners (bpsplit and bpappend).
    b. Archivers
    c. File Transfers: scp and sftp
    d. Visualization tools: Xmgrace [80] and AVS [81].

We also ran monitor modules on these machines to check the status of the services above and report back to the error-state handling layer. Note that when possible, functionally equivalent services were used on each of the Red Hat Servers, for example Xmgrace, and AVS, in order to minimize error correlation among contingencies. The machines were interconnected using the campus network, and network outages were not part of the assessment scenarios.

Figure 8.2 shows the workflow that ran on the Windows XP machine (laptop). This workflow is a simplified version of the XGC and S3D production workflows, but does include common operations such as file monitoring, file transfer, data conversion, data archiving and visualization found in the ORNL versions of the workflows. It is a serial workflow, which means all actors are executed in a serial manner, even though the execution

flow after the “Convert” actor splits into 2 branches. In fact, if run with PN director, all actors are started simultaneously (in parallel) but have blocking input queues that wait for appropriate inputs to arrive before they actually execute. The “Archiver” actor is non-essential and does not affect the success of the overall workflow run. So the main execution flow is “reduced” to 6 serial actors, which is similar to the serial nature of production workflows. As we mentioned in section 2.4.1 when presenting the XGC and S3D production workflows, production workflows might appear to have parallel execution flows, but effectively they are serial flows, since the execution eventually blocks waiting on all the previous actors to finish before proceeding further. Therefore, we believe that POC and the test workflow offer a reasonable environment for initial evaluation of the properties of the proposed fault-tolerance framework.

The actors marked with a “C” are contingency actors (File Transfer, Convert and Create Image Actors), and each of them has its prime middleware service running on the first RHEL Server, and backup middleware service (alternative) running on the second RHEL Server. Following is a description of the actors:

SimLauncher: A stateful actor, responsible for the invocations of the simulation code. The supercomputing application simulation code will generate simulated output data (.bp files) for each invocation based on the parameters supplied by the SimLauncher actor.

FileTransfer: A stateless actor, responsible for transfering the simulation data files from the simulation machine (“supercomputer”) to the analysis machine.

Convert: A stateless actor, responsible for converting the simulation data files from bp format (.bp) to hdf5 format (.h5).

FilenameFilter: A stateless actor, responsible for preparing the simulation data files for the vizualization sevices.

CreateImage: A stateless actor, responsible for launching the vizualization services and supplying the hdf5 data files as input.

ShowRemoteImage: A stateless actor, responsible for displaying the images and movies produced by the vizualization services.

Archiver: A stateless actor, responsible for archiving the simulation data files.

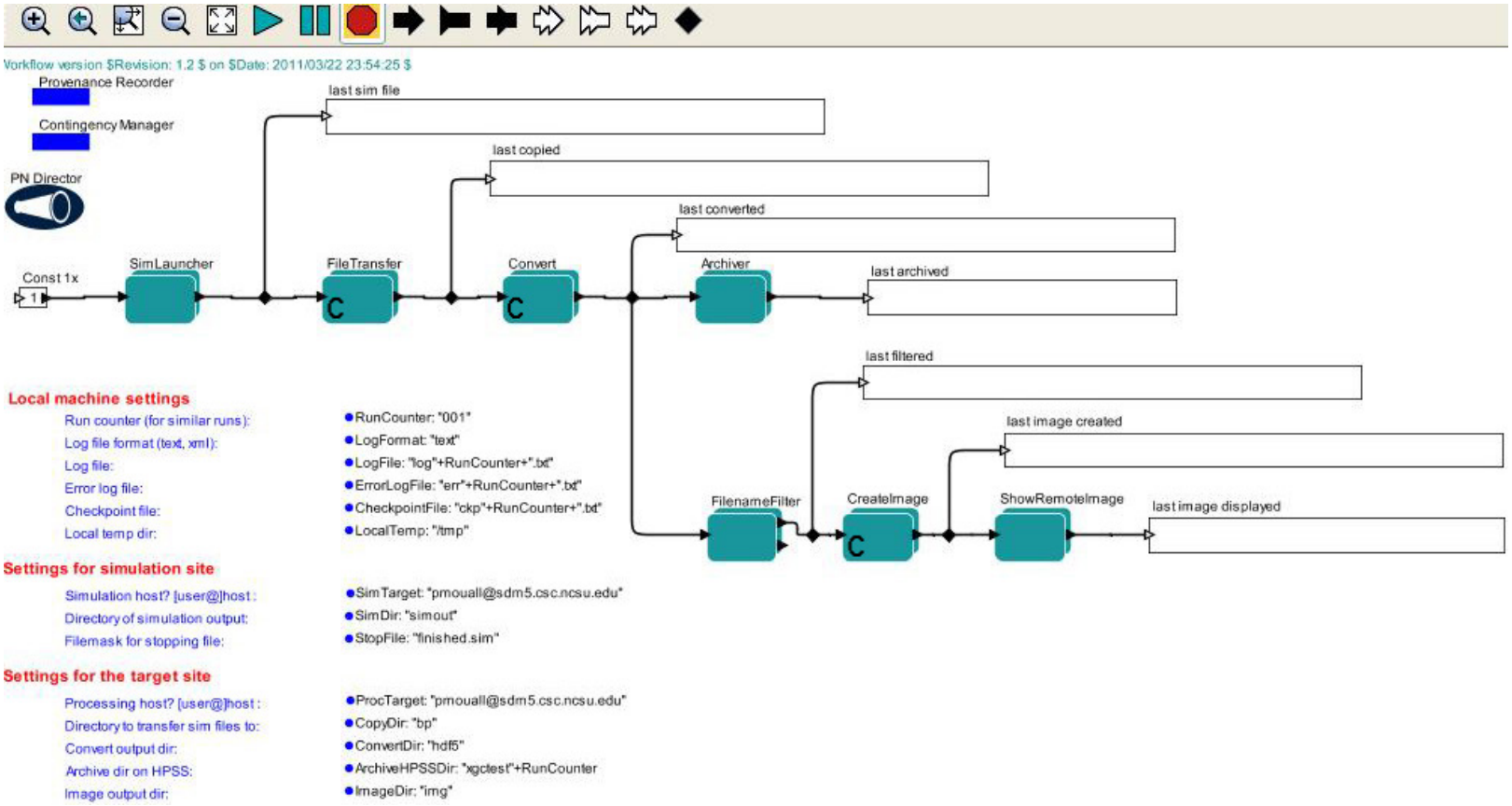

Figure 8.2: Test case Workflow

8.2 Reliability evaluation

We start the evaluation by assigning failure probabilities (expressed ad percentages in the graphs and tables that follow) to the components of the test case workflow (figure 8.2). These probabilities are based on the estimated failure probabilities of equivalent processes in production workflows as presented in table 2.1. Failure injection is effected using wrapper scripts for the emulated/simulated middleware services, and by modifying the actors so that they fail randomly using the assigned failure probability. The workflow actors call those wrappers, which in turn call the services and components the actors need.

The “File Transfer”, “Convert” and “CreateImage” actors are in large part dependant on the underlying middleware and operating systems. Therefore they have been assigned different failures probabilities that are similar to the failures probabilities expected for the middleware and OS layers on which they depend and similar to what was observed in the case of the XGC and S3D production workflows. For the rest of the actors, such as “SimLauncher”, FilenameFilter”, “ShowRemoteImage”, we assume they all have the same failure probability as actors of production workflows that are not dependant on other middleware services or components.

To assess consistency of the theoretically expected and actual behavior of the proof-of-concept implementation and we ran the following three scenarios.

Scenario 1: *No contingency actors used, error-state handling layer disabled*

This is the situation which simulates a workflow that is not assisted by the proposed framework. We start by executing the workflow of figure 8.2 without using any contingency

actors and by disabling the error-state handling layers. For a serial workflow to be successful each actor needs to succeed. So in the case of the workflow of figure 8.2, all 6 actors need to succeed (note that the seventh actor, "Archiver", is on a side-branch that is not considered crucial for completion of the workflow and therefore is assumed NOT to result in the failure of the workflow even if the actor fails). Therefore, **assuming** there is no failure correlation among the actors, the probability of a workflow run failing can be calculated using the following equation [10] [82].

$$P_F = 1 - \prod_{i=1}^{n} (1 - P_i) \quad (1)$$

Where P_F is the probability that the workflow fails, P_i is the probability that actor i fails, and n is the number of actors in the workflow. As already mentioned, in this scenario we assume that actors are in series, n = 6, their failures are independent, and that no correlation exists among failures.

We now inject failures into the workflow components of figure 8.2. This is done by having the expected failure probability values saved into a text file which the workflow launcher reads and assigns to the wrappers of individual actors. The wrappers, using a random number generator, then determine whether the actor should fail or continue execution. Figure 8.3 below shows the distribution of the failure probabilities used in one of the experiments – the one that attempted to mimic observed production ratios among the failures (table 2.1). Each instance (horizontal axis) represents four failure probability values – one for the four different types of actors, and reflecting different relative failure behavior of the middleware, O/S, hardware and control layers. On the vertical axis is the failure probability (expressed as

percentage) that was used to randomly inject failures at each instance. Each instance consisted of run 100 simulated runs.

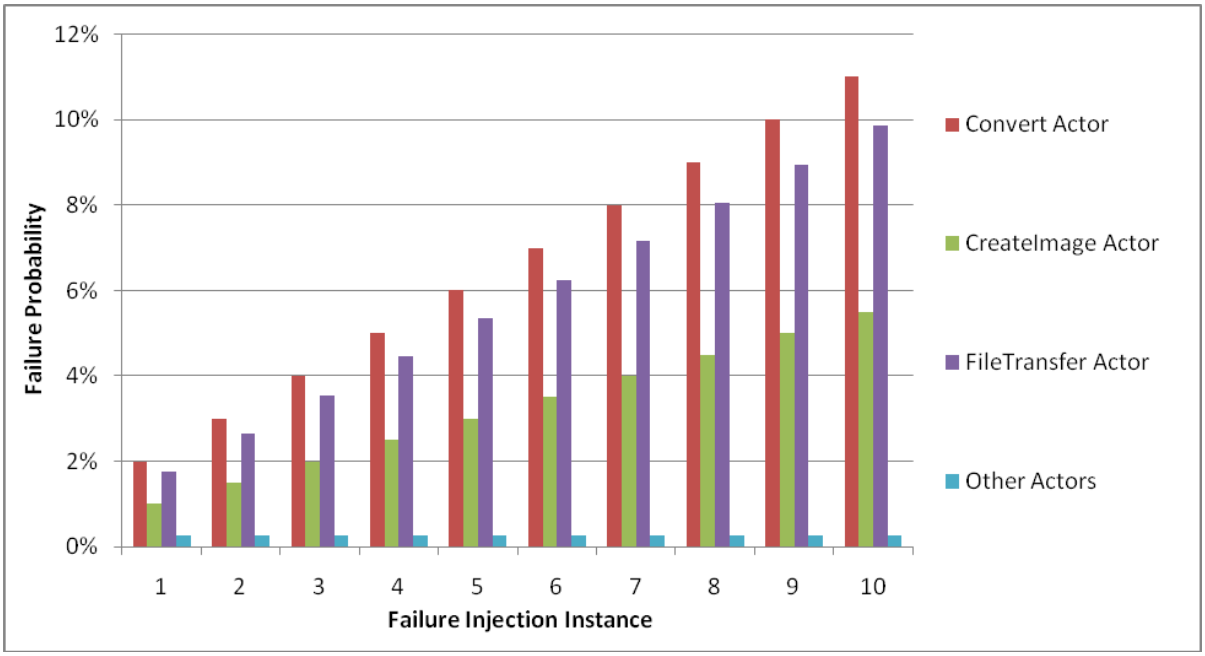

Figure 8.3: Failure probabilities used for individual components in ten experiments.

For example, let's consider instance 1. In this instance, a failure probability of 0.02 is assigned to the "Convert" actor, a failure probability of 0.0175 is assigned to the "FileTransfer" actor, and a failure probability of 0.01 is assigned to the "CreateImage" actor. The remaining actors (SimLauncher, FilenameFilter, ShowRemoteImage) are assumed to have failure probability of only 0.0025, similar to the failure probability of actors in the production workflows that do not depend on middleware services or components. As mentioned earlier, the reason that the first 3 actor types have higher failure probabilities is because they depend on the middleware components and are more susceptible to the failures

occurring at middleware and OS layer. The value ratios of figure 8.3 correspond to the failure percentages of similar actors/services in the production workflows.

Figure 8.4 shows the results of the simulation runs (orange bars – second bar in each bar graph). Based on the injected failures (figure 8.3), 1000 trial runs for instance 5, and 100 trial runs for the remaining instances, the overall estimated probability that a workflow run fails, is shown on the y-axis (and again expressed as percentage). Although we would have preferred to have 1000 trial runs for each instance, it was not feasible in the context of this dissertation due to time and hardware limitations. However having at least one instance with 1000 trial runs helps us spot check the accuracy of the improvements and conclusions. This is observed in the figure below where the results for scenario 5 (1000 runs) are consistent with the results of other scenarios (100 runs), thus confirming the validity of the results.

We also calculated the theoretical failure probability of the workflow as a whole using equation (1) as applied to the workflow of figure 8.2. This is shown in figure 8.4 as the blue (left-hand) bar.

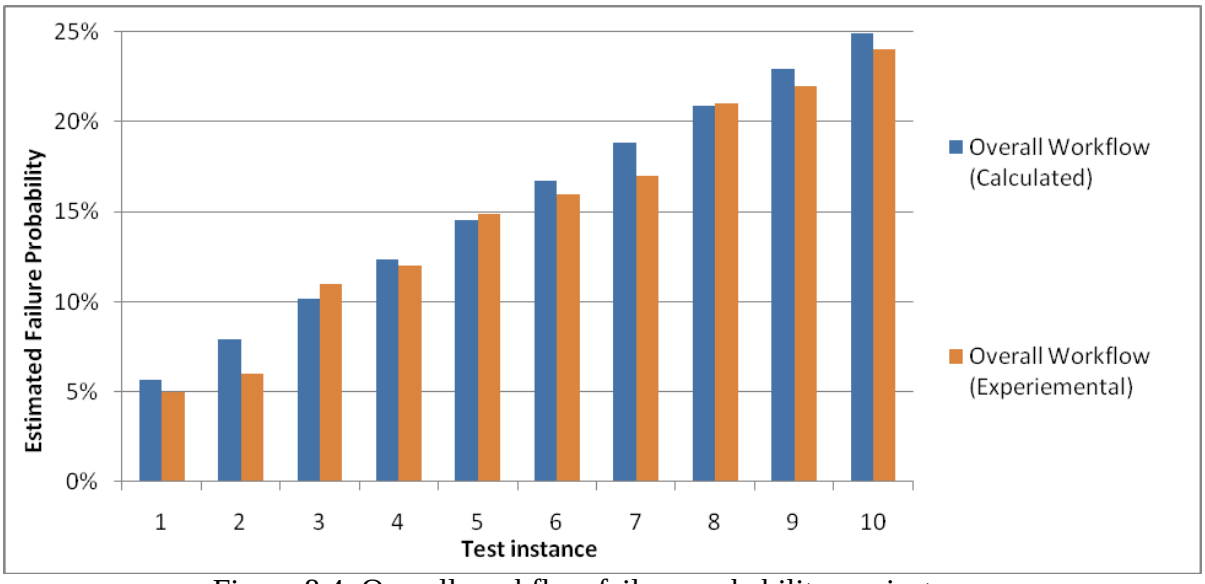

Figure 8.4: Overall workflow failure probability per instance

This difference in figure 8.4 between the experimental results and theoretical results are minimal and fall within the simulation error bounds, thus confirming the consistency of the presented model. The orange bars are generally lower due to the limited number of simulation runs executed for each case (100 runs), which does not accurately reflect the failure probabilities of actors 1, 4 and 6 since it is much lower (0.0025). But as we mentioned they still fall within the error bounds.

Figure 8.5 shows the runtime of 100 runs for instances 1, 4, 7 and 10 (the remaining instances were not displayed so that the graph would remain legible).

Figure 8.5: Runtime per run/instance

On average, a successful run take around 9 minutes (black line). The runtimes below 450 seconds represent failed runs. Note that in this case, failed run durations are always less than the successful run durations because we are injecting failure and no timeouts or propagated failures are occurring. That is not the case in production workflows, and the runtime of failed runs can be higher than the runtime of successful runs.

The average runtime per successful run is calculated by taking the overall runtime of the 100 runs (or 1000 runs for scenario 5) and dividing it by the number of successful runs. This is represented in figure 8.6 below. Again we notice that the results of scenario 5 are consistent with the results of other scenarios, thus further validating those results.

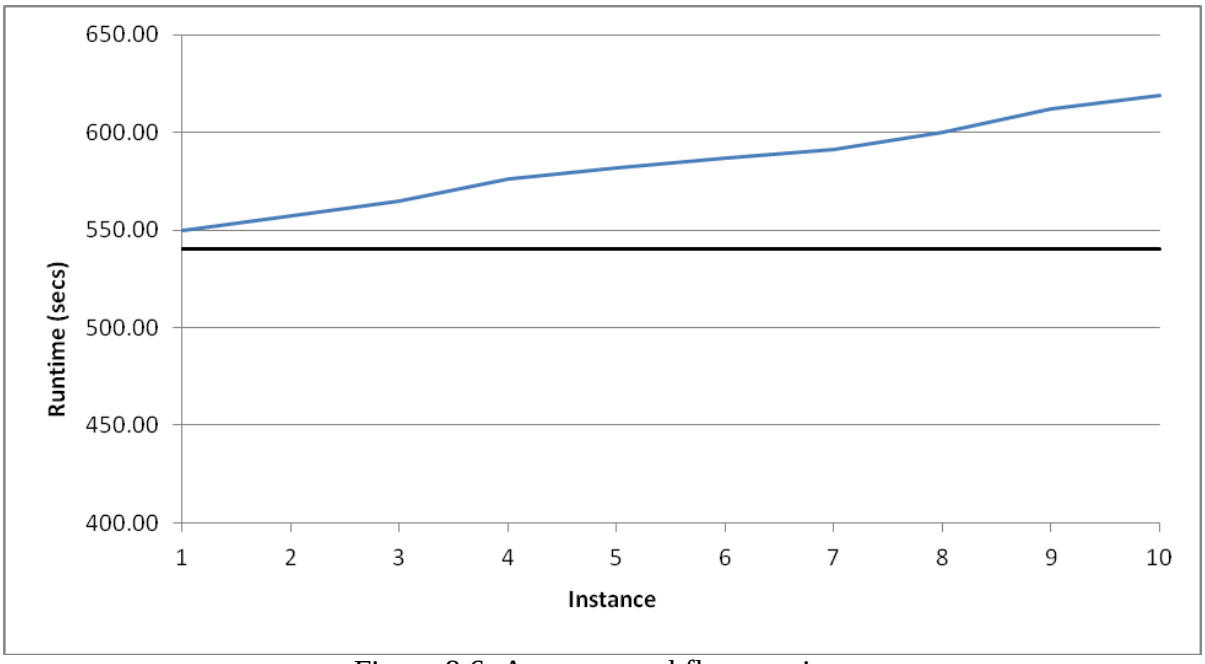

Figure 8.6: Average workflow runtime

The black line represents the average runtime when no failures occur, and the blue line represents the average runtime when failures are possible. As expected, the higher the failure probability, the higher the average run time per successful run. At instance 10, where the failure probability is 0.24, the average run time per run is 620 seconds, 80 seconds higher than the average run time when no failures occur, which represents a 14.8% time increase. Note that when failed runs are factored in, the average runtime depends on the duration of those failed runs and whether they fail early or late. As mentioned above, the failed runs in our test cases have a shorter duration that successful runs since we are injecting failures and no timeouts or propagated failures are occurring. That is not the case in production workflows which would result in a higher time increase than what is shown in figure 8.6.

Scenario 2: *Contingency actors used, error-state handling layer disabled*

In this scenario we activate only the workflow control layer resident fault-tolerance mechanisms. We assess fault tolerance using the contingency actor and its alternatives, but without any help from the error-state handling layer. The impact of the latter is discussed below. As mentioned in section 8.1, we have three different types of contingency actors (marked with a "C"), and each contingency actor has a prime and a backup service, each of them running on one of the Red Hat Enterprise 5 servers. When possible, we used functionally equivalent services, such as AVS and XM-Grace, each running on a separate server, in order to reduce failure correlation, otherwise we ran the same service on both servers.

In general, there could be more than one alternative for each actor. For example, one could have three functionally equivalent graphics services. Workflow would fail if the prime (say AVS) and both alternatives failed. This can happen, for example, if machines hosting these services fail, licenses have expired, or network is down. Without error-handling layer, the actor in question would first try the primary service, then the secondary and then the tertiary. However, if the issues disabled all three alternatives, the error handling layer could short-circuit the process and allow the workflow to immediately initiate another recovery option (e.g., checkpoint rollback), instead of wasting time by trying out failed services. Some solutions suggest [72] that alternatives need to be run in parallel – if that option is available – to reduce the serialization overhead, but that can also have time-out delays and similar issues.

For a contingency actor to succeed, at least one of its contingencies needs to be successful. So we can calculate the failure probability of an actor with multiple contingencies using the following equation:

$$P_{Fi} = \prod_{j=1}^{m} p_j \quad (2)$$

Where p_j is the failure probability of contingency j, and m is the number of contingencies.

Applying equation (2) to equation (1), we can calculate the failure probability of the entire workflow using the following equation. Note that this equation assumes that all failures (inter-actor and inter-contingency) are independent:

$$P_F = 1 - \prod_{i=1}^{n} (1 - \prod_{j=1}^{m} p_{ij}) \quad (3)$$

P_F is the probability that the workflow fails, p_{ij} the probability that contingency j of actor i fails, n the number of actor in the workflow, and m the number of alternatives. Note that for the actors that are not contingency actors, m is equal to 1.

Our basic assumption in equation (2) is that there is no failure correlation. Failure probabilities of the alternative services are independent of each other. However, when that is not the case, we consider workflow failure probability when failure correlation among contingencies is possible. For illustration, let us assume that each actor has only two contingencies (prime and backup), and that failures among actors are independent. This, of course may not be realistic if all actor initiated services are running on one machine, and there are issues with the hardware on which these services are running.

When failure correlation exists between two services, say A1 and A2, the probability of failure of each becomes a conditional probability, represented with the following equation [82]:

$$P(A_1 \cap A_2) = P(A_1)P(A_2 \mid A_1) \quad (4)$$

Where A_1 represents contingency 1 of actor A, A_2 represents contingency 2 of actor A, $P(A_1 \cap A_2)$ represents the failure probability of A_1 and A_2. $P(A_1)$ represents the failure probability of A_1, and $P(A_2 \mid A_1)$ represents the probability that A_2 will fail given that A_1 has failed. This basically means that if the failure probability of A2 is measured on its own, it will be different than if it is measured knowing that A1 has already failed.

Applying (4) to (1), we calculate the failure probability of the entire workflow using the following equation:

$$P_F = 1 - \prod_{i=1}^{n} (1 - P(A_{i1} \cap A_{i2})) \quad (5)$$

Where P_F is the probability that the workflow fails, n is the number of actors in the workflow, $P(A_{i1})$ the probability that contingency 1 of actor i fails, $P(A_{i2})$ the probability that contingency 2 of actor i fails. Note that when failure correlation is equal to 0, this means that contingencies failures are independent events and that $P(A_1 \cap A_2) = P(A_1)P(A_2)$, resulting in (5) being the same as (3). And when failure correlation is at 100%, meaning that if contingency 1 fails, contingency 2 has a 100% chance of failing, this makes $P(A_1 \cap A_2) = P(A_1)$, thus resulting in (5) being equal to (1).

Using (3) and (5), we calculate the failure probability of a workflow given the failure probabilities of the individual components in figure 8.3. Figure 8.7 shows the results.

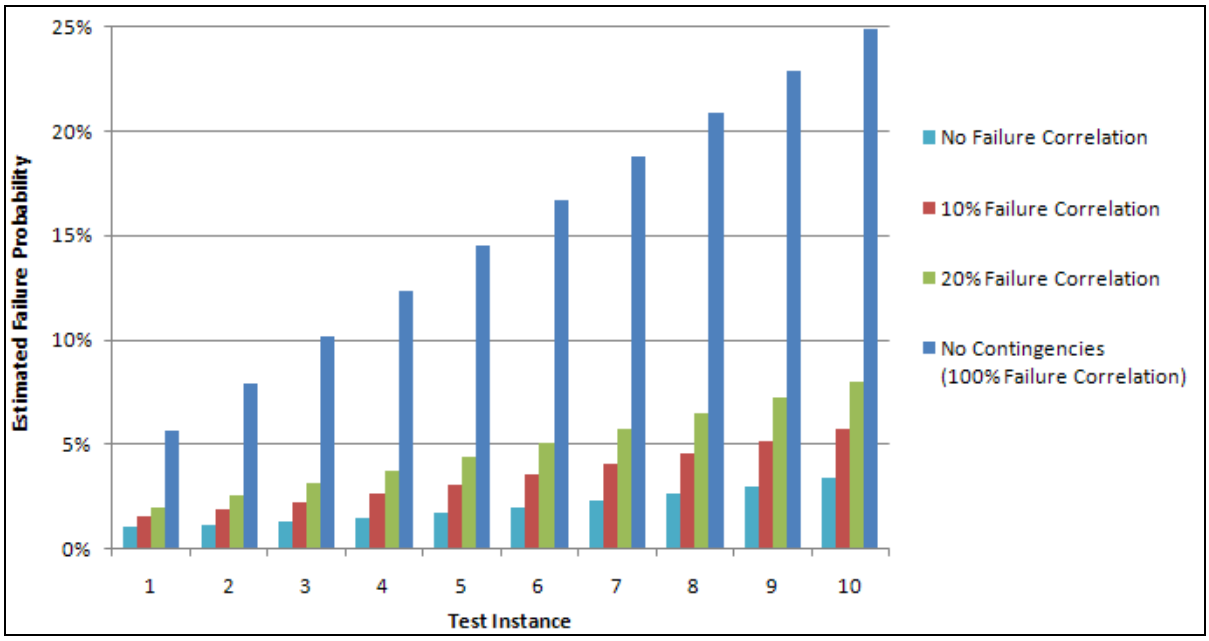

Figure 8.7: Calculated failure probability of test workflow with contingency actors (scenario 2)

Figure 8.7 shows the calculated failure probabilities of 4 scenarios of failure correlation for each of the test instances shown in figure 8.3. The overall estimated probability that a workflow run fails, is show on the y-axis (expressed as percentage). We calculated the theoretical failure probability of the workflow as a whole using equation (5) as applied to the workflow of figure 8.2. The worst case scenario, when failure correlation is at 100%, is equivalent to not having any contingencies (dark blue bars), since if the first contingency fails, all resulting contingencies will fail as well. The best case scenario, when there is no failure correlation (light blue bars), is equivalent to the independent model shown in equation

(3). The red and green bars represent failure correlation probabilities of 10% and 20% respectively. As expected, the higher the failure correlation probability gets, the higher the overall workflow failure probability becomes.

Figure 8.8 shows the experimental failure probabilities of 1000 runs for instance 5 and 100 runs for the remaining instances of figure 8.3 for each of the 4 scenarios of correlation per test instance. Having at least one 1000 run test case helps us verify the accuracy of the presented model since it would be more sensitive to the low failure probabilities of the non-contingency actors. It also helps us validate the test cases with 100 runs by comparing the consistency of the results of those test cases with the results of the test case with 1000 runs.

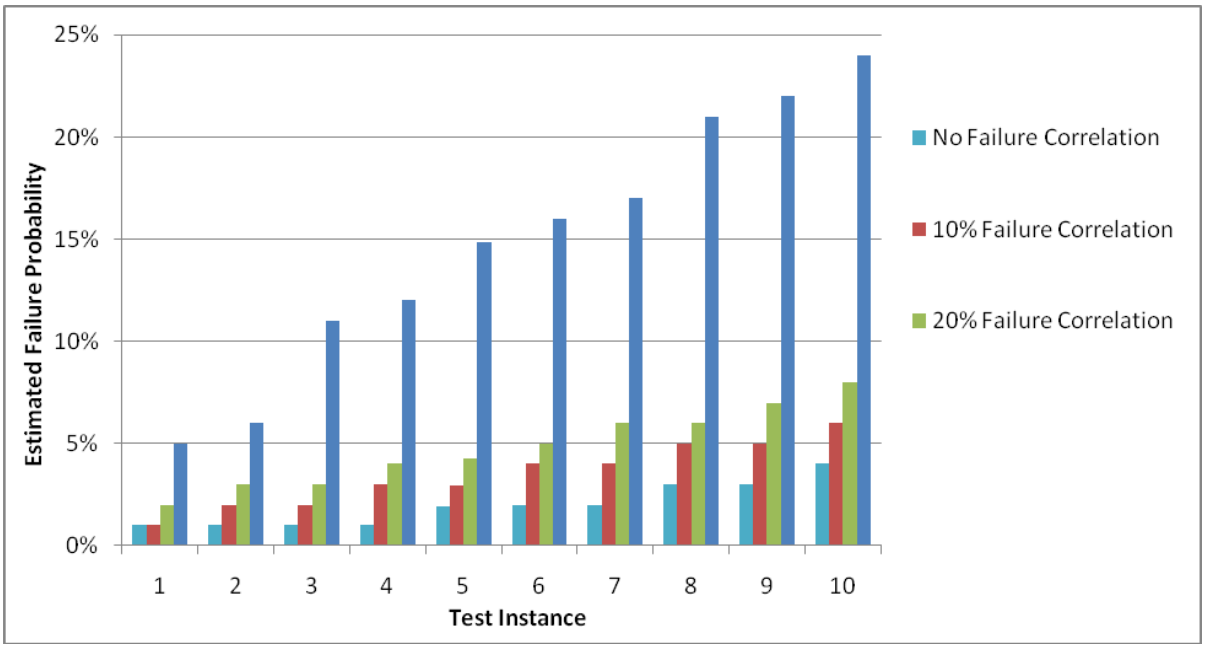

Figure 8.8: Experimental failure percentage of test workflow with contingency actors (scenario 2)

Note that the accuracy of simulations shown here is relatively low for the instances with 100 runs, however the numbers we obtained are a good indicator of the difference between the correlation levels.

Figure 8.9 shows the runtime of 100 runs for instances 1, 4, 7 and 10 (the remaining instances were not displayed so that the graph would remain legible) when no failure correlation is possible. Since the contingency actor introduces retries based on failure/timeout of contingency, the duration of some of the successful runs is significantly higher than the average run time of 9 minutes shown in figure 8.5. Furthermore certain failed runs have a higher than average run time based on the number of retries that occurred.

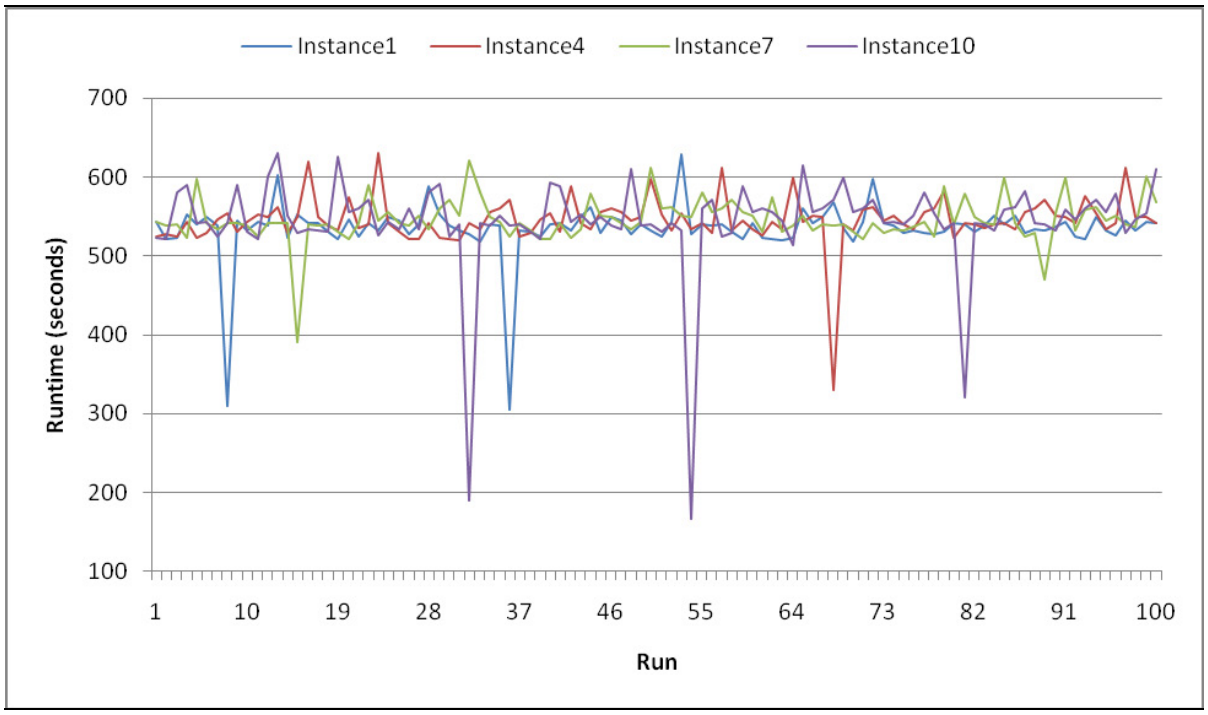

Figure 8.9: Runtime per run/instance using contingencies with no error correlation (scenario 2)

Notice that the runtimes of successful runs display a higher level of fluctuation for when compared to the runs when no contingency actors were used (figure 8.5). This is due to the retry/timeout mechanism of the contingency actor (each contingency time out after 30 seconds of irresponsiveness). However, the number of failed runs is significantly smaller. Table 8.1 lists the runs in which failures occurred. It also lists the contingencies that were executed and the outcome of the runs. Note that the prime contingency will always be executed first, and only when it fails does the backup contingency get executed.

Table 8.1: Runs of instance 1 in which failures occurred (scenario 2)

	Actor / Run #	1	2		3		4	5		6	Outcome
			Prime	Backup	Prime	Backup		Prime	Backup		
Instance 1	13	✓	✓	---	✗	✓	✓	✗	✓	✓	Success
	28	✓	✗	✓	✓	---	✓	✓	---	✓	Success
	36	✓	✓	---	✗	✗	---	---	---	---	Failure
	53	✓	✗	✓	✓	---	✓	✓	---	✓	Success
	72	✓	✓	---	✓	---	✓	✗	✓	✓	Success
Instance 4	16	✓	✗	✓	✓	---	✓	✓	---	✓	Success
	20	✓	✓	---	✓	---	✓	✗	✓	✓	Success
	23	✓	✓	---	✗	✓	✓	✗	✓	✓	Success
	42	✓	✗	✓	✓	---	✓	✓	---	✓	Success
	50	✓	✓	---	✓	---	✓	✗	✓	✓	Success
	57	✓	✓	---	✗	✓	✓	✓	---	✓	Success
	64	✓	✗	✓	✓	---	✓	✓	---	✓	Success
	68	✓	✓	---	✗	✗	---	---	---	---	Failure
	79	✓	✗	✓	✓	---	✓	✓	---	✓	Success
	93	✓	✗	✓	✓	---	✓	✓	---	✓	Success
	97	✓	✓	---	✗	✓	✓	✓	---	✓	Success
Instance 7	5	✓	✗	✓	✓	---	✓	✓	---	✓	Success
	15	✓	✓	---	✗	✗	---	---	---	---	Failure
	22	✓	✗	✓	✓	---	✓	---	---	✓	Success
	32	✓	✓	---	✗	✓	✓	✗	✓	✓	Success
	33	✓	✓	---	✓	---	✓	✗	✓	✓	Success
	44	✓	✗	✓	✓	---	✓	✓	---	✓	Success

Table 8.1 Continued

	50	✓	✗	✓	✗	✓	✓	✓	---	✓	Success
	55	✓	✓	---	✗	✓	✓	✓	---	✓	Success
	62	✓	✓	---	✓	---	✓	✗	✓	✓	Success
	79	✓	✓	---	✗	✓	✓	✓	---	✓	Success
	81	✓	✗	✓	✓	---	✓	✓	---	✓	Success
	85	✓	✗	✓	✓	---	✓	✓	---	✓	Success
	89	✓	✓	---	✓	---	✓	✗	✗	---	Failure
	91	✓	✓	---	✗	✓	✓	✓	---	✓	Success
	99	✓	✗	✓	✓	---	✓	✓	---	✓	Success
Instance 10	3	✓	✓	---	✗	✓	✓	✓	---	✓	Success
	4	✓	✗	✓	✓	---	✓	✓	---	✓	Success
	9	✓	✓	---	✓	---	✓	✗	✓	✓	Success
	12	✓	✗	✓	✓	---	✓	✓	---	✓	Success
	13	✓	✓	---	✓	---	✓	✗	✓	✓	Success
	19	✓	✗	✓	✓	---	✓	✗	✗	---	Failure
	28	✓	✓	---	✗	✓	✓	✓	---	✓	Success
	29	✓	✓	---	✗	✓	✓	✓	---	✓	Success
	32	✓	✗	✗	---	---	---	---	---	---	Failure
	40	✓	✓	---	✓	---	✓	✗	✓	✓	Success
	41	✓	✓	---	✗	✓	✓	✓	---	✓	Success
	48	✓	✗	✓	✓	---	✓	✓	---	✓	Success
	54	✓	✗	✗	---	---	---	---	---	---	Failure
	59	✓	✗	✓	✗	✓	✓	✓	---	✓	Success
	65	✓	✓	---	✗	✓	✓	✓	---	✓	Success
	69	✓	✓	---	✗	✓	✓	✓	---	✓	Success
	77	✓	✓	---	✗	✓	✓	✓	---	✓	Success
	81	✓	✓	---	✓	---	✓	✗	✗	---	Failure
	87	✓	✗	✓	✓	---	✓	✓	---	✓	Success
	96	✓	✓	---	✗	✓	✓	✓	---	✓	Success
	100	✓	✗	✓	✗	✓	✓	✓	---	✓	Success

Note that actors 1, 4 and 6 have a very lower failure probability and did not fail in the experimental runs performed. If the number of runs were to increase (up to a 1000 or more), the granularity of this test case would be finer and we would see example where those actors might fail. However executing that many runs for all the scenarios/instances would require much larger computing resources and time than what is currently available.

Figure 8.10 shows the runtime of 100 runs for instances 1, 4, 7 and 10 (the remaining instances were not displayed so that the graph would remain legible) when failure correlation is equal to 20%. As we have seen in figure 8.9, the runtimes fluctuations remain high due to the introduction of the contingency actor.

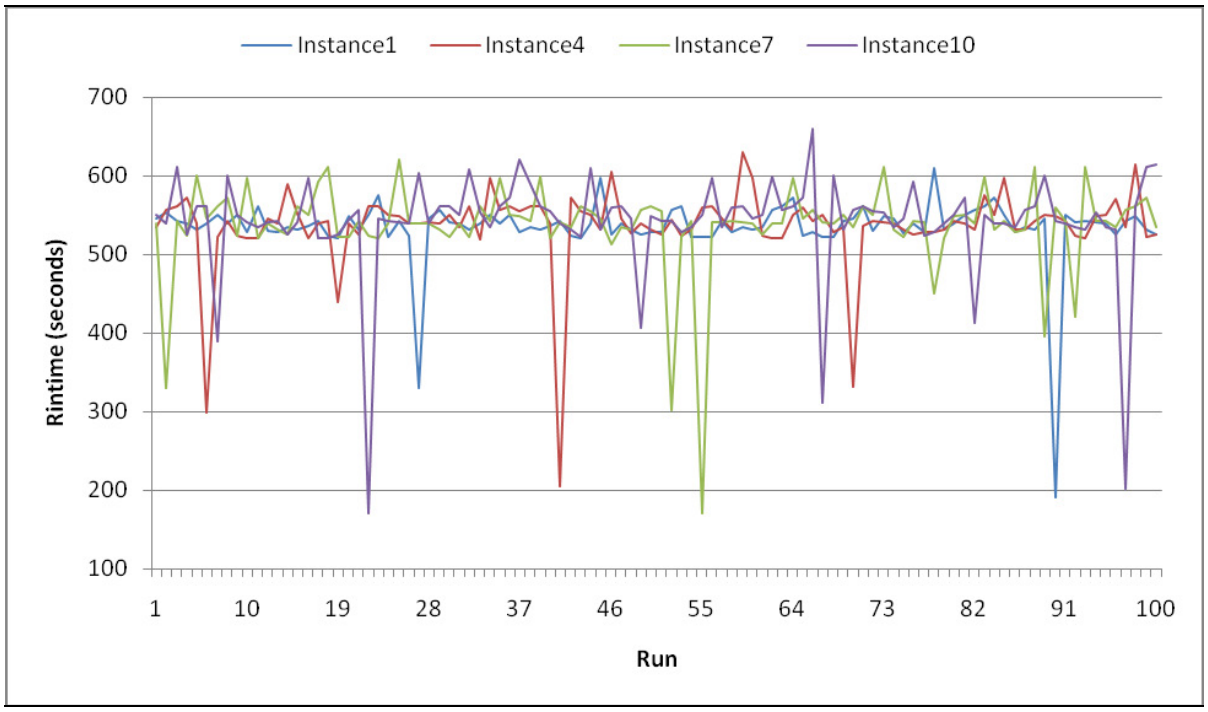

Figure 8.10: Runtime per run/instance using contingencies with 20% failure correlation (scenario 2)

Notice that the runtimes of successful runs display a higher level of fluctuation when compared to the runs where no contingency actors were used (figure 8.5). However the number of failed runs for this scenario is much lower. Yet when compared to the case where contingency actors with no failure correlation was possible (figure 8.9), the number of failed runs is higher.

Figure 8.11 shows the average runtime of a successful run (per instance) for the following scenarios of figure 8.7: No failure correlation, 20% failure correlation and no contingencies. The average runtime per successful run is calculated by taking the overall runtime of a 100 runs and dividing it by the number of successful runs.

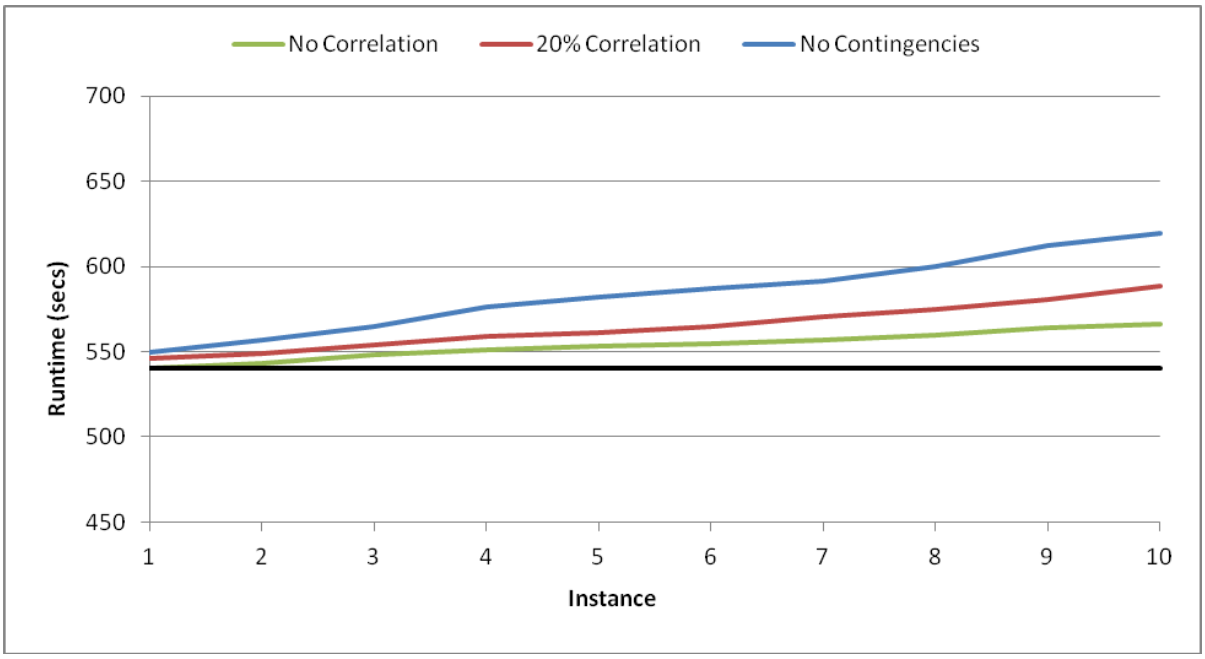

Figure 8.11: Average runtime of a successful run (scenario 2)

Although the use of the contingency actors significantly lowers the workflow failure probability as seen in figure 8.8, this comes at the price of an increased runtime as shown in figure 8.11, especially when the failure probability of individual components goes up. At instance 10, the average run time when no error correlation occurs (green line) is 562 seconds, which represents a 4.1% time increase over the average run time of a successful run when no errors occur (black line). If we consider the case when error correlation is at 20%

(red line), the average run time is 589 seconds, which represents a 9.1% increase over the average run time of a successful run when no errors occur (black line). Scenario 3, which is presented below, aims to address the increases in runtimes even when failure correlation is possible.

Scenario 3: *Contingency actors used, error-state handling layer and checkpointing mechanism enabled*

In this scenario, we run the test workflow with all components of the presented fault tolerance framework. This includes workflow-level actor-based fault-tolerance, error-state management, and roll-back and recovery via checkpointing. The actor-based recovery is a forward-recovery technique and masks failures. The checkpointing mechanism is a backward recovery mechanism and while it does not directly improves the success rate of a single workflow run, its primary function is cost management through reduction of the runtime of re-executions that need to take place in case of failed workflow runs. In turn, this reduces both the overall workflow end-to-end run time as well as CPU time used.

Similarly error-state management provides additional information on failures that otherwise would either fail the workflow completely, or at best cause a partial or full roll-back and re-execution. It allows fine granularity tuning of the workflow level responses to failures. This allows better problem and provenance logging, better failure and fault diagnostics, and can help avoid unnecessary actor and workflow re-executions.

To test this scenario, the workflow of figure 8.2 is used with a modification to the contingency actor. Instead of having a prime contingency that always gets executed first, and a backup contingency that gets executed when the prime fails, we now have 3 contingencies as follows:

1. **Suggested Contingency:** The error handling layer notifies the contingency actor at runtime which underlying components/services with the highest probability of success to use, based on the error taxonomy/analysis of previous runs. The underlying components/services are selected when the workflow is launched. During the workflow execution, if the error-state handling layer determines that the selected components/services availability has deteriorated and that other components/services are a better alternative, the contingency actor is notified of the changes and new components/services are assigned for use.
2. **Default Contingency:** It is executed by default and decided pre-runtime. It is required to allow the workflow execution to proceed in case the error-state handling layer becomes unavailable during execution (network outage, hardware error, etc), and would not be able to provide the contingency actor with input on what contingency to choose. Without this option, the contingency actor would wait on input and eventually time out causing the workflow to fail. Note that the suggested contingency can be the same as the default contingency, in which case the default contingency is used, and if the default contingency fails, the error-state handling layer is notified and, if available, a different underlying component/service is suggested during runtime, otherwise the abort contingency is used executed.

3. **Abort Contingency:** It is used to gracefully stop the workflow execution if all the other contingencies were determined by the error handling layer to be unavailable or expected to fail, in which case the error-handling layer would re-run the workflow using the last successful checkpoint (if it has been setup to do so by the user). Without a graceful exit, the workflow might continue running for a period of time before halting, thus wasting time and resources, as seen in the failed workflow run example of Appendix A.2, in which case the workflow continued running for 42 additional minutes after the first error occurring at minute 11.

Note that the execution order of these contingencies is decided via input (or lack of) from the error-handling layer. Also for our testing purposes, we limit the number of alternatives per service to one, each running on one of the RHEL servers.

The probability of a workflow run failing when checkpointing is enabled can be calculated by taking the probability of one run failing which is represented by equation (5), and multiplying it by the number of checkpointing based retries (resumes). This assumes that the failures between re-executions are not correlated. For example, that the failures are not due to a workflow or actor design error. For this scenario, we also make the same assumptions used for scenario 2 (two contingencies per contingency actor, uncorrelated failures between actors). The probability of failure can be calculated with the following equation:

$$P_F = \prod_{k=1}^{r} (1 - \prod_{i=1}^{n} (1 - P(A_{i1} \cap A_{i2}))) \quad (6)$$

Where P_F is the probability that the workflow will fail, n is the number of actors in the workflow, $P(A_{i1})$ the probability that contingency 1 of actor i fails, $P(A_{i2})$ the probability that contingency 2 of actor i fails and r the number of checkpointing based re-runs. Note that equation (6) represents a worst case scenario when it comes to checkpointing based workflow re-run, since it assumes that the entire workflow is being re-executed, however in reality the actors that executed successfully and were included in the latest checkpoint will not be re-executed, thus reducing the failure probability of the workflow re-run/resume.

Using (6), we calculate the failure probability of a workflow given the failure probabilities of the individual components in figure 8.3. The "abort" contingency is configured to resume execution once before giving up. Figure 8.7 shows the results.

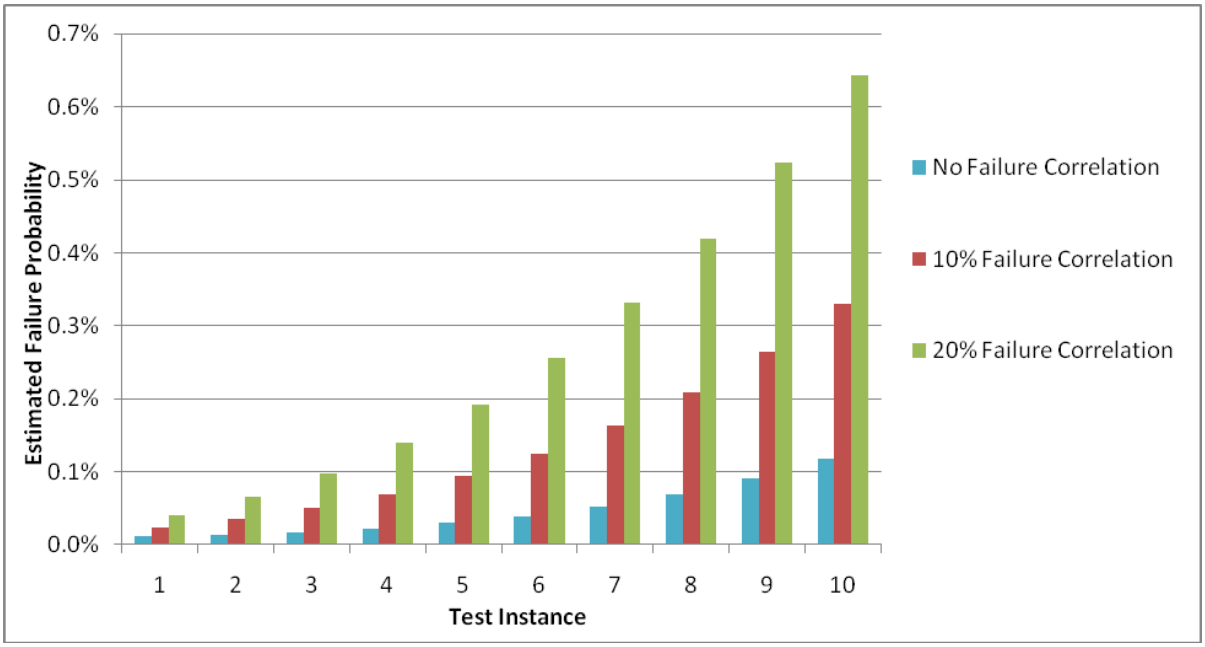

Figure 8.12: Calculated failure probability of test workflow with contingency actors (scenario 3)

Figure 8.12 shows the calculated failure probabilities of 3 scenarios of failure correlation for each of the test instances shown in figure 8.3. The overall estimated probability that a workflow run fails, is show on the y-axis (expressed as percentage). When comparing these results to the results of the same test in scenario 2, the failure probabilities in this scenario are reduced by a factor of 10 or more.

Calculating the failure probability for this scenario using the test workflow would require a very large number of runs (10^4 or more to get any significant result) which currently is not viable in the context of this dissertation. Therefore we shall rely on the calculated results for comparison purposes. However we do run the test workflow for this scenario to capture the runtimes and compare the results with the runtimes of scenario 2.

We execute the test workflow using the same failure probabilities of individual components shown in figure 8.3, and for the 3 scenarios of failure correlation shown in figure 8.7 (no failure correlation, 10% failure correlation and 20% failure correlation). We also configure the “abort” contingency to resume execution once before giving up. Figure 8.13 shows the runtime of 100 runs for instances 1, 4, 7 and 10 when no failure correlation is possible.

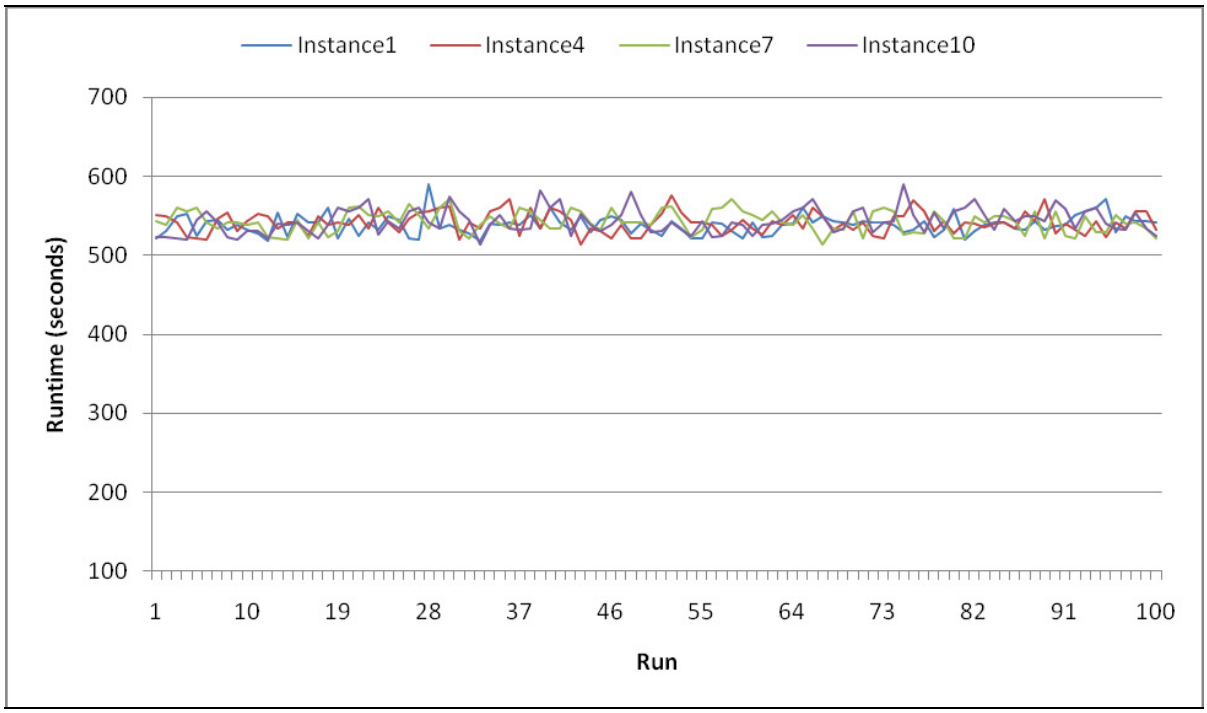

Figure 8.13: Runtime per run/instance with no error correlation (scenario 3)

The runtimes of successful runs shown above display a much lower level of fluctuation (variance) in this scenario when compared to the same test in scenario 2 (figure 8.9). This is due to the error-state handling layer actively instructing the workflow on which components/services to use, thus drastically reducing the amount of retries. This is shown in table 8.2, which lists the runs in which components/service were determined to be unavailable. It also lists sample runs in which different contingencies were executed and the outcome of the runs. For example, in instance 1 run 32 from table 8.2 below, the workflow executed the services suggested by the error-state handling layer and no failures were encountered in the execution. In instance 1 run 18, the error-state handling layer determined that no viable services for actor 3 were available, so it instructed the workflow to execute the

abort contingency (run 1), and use the last successful checkpoint to resume the workflow (run 2). In this case, a minimal amount of runtime was wasted since the workflow did not have to wait on services to timeout before aborting the run.

Table 8.2: Runs of instance 1 in which failures occurred (scenario 3)

	Actor / Run #		1	2			3			4	5			6	Outcome
				Suggested	Default	Abort	Suggested	Default	Abort		Suggested	Default	Abort		
Instance 1	18	pass 1	✓	✓	---	---	---	---	✓						Success
		pass 2					✓	---	---	✓	✓	---	---	✓	
	32		✓	✓	---	---	✓	---	---	✓	✓	---	---	✓	Success
	78		✓	---	✓	---	---	✓	---	---	✓	---	---	---	Success
Instance 4	16		✓	✓	---	---	✓	---	---	✓	✓	---	---	✓	Success
	20		✓	✓	---	---	✓	---	---	✓	---	✓	---	✓	Success
	52	pass 1	✓	✓	---	---	✓	---	---	✓	---	---	✓		Success
		pass 2									✓	---	---	✓	
	88	pass 1	✓	---	---	✓									Success
		pass 2		✓	---	---	✓	---	---	✓	---	---	✓	✓	
Instance 7	26	pass 1	✓	---	---	✓									Success
		pass 2		✓	---	---	✓	---	---	✓	---	---	✓	✓	
	46	pass 1	✓	✓	---	---	✓	---	---	---	---	---	✓		Success
		pass 2									✓	---	---	✓	
	61		✓	✓	---	---	✓	---	---	✓	✓	---	---	✓	Success
	84		✓	✓	---	---	---	✓	---	✓	---	✓	---	✓	Success
Instance 10	6	pass 1	✓	---	✗	✓									Success
		pass 2		✓	---	---	✓	---	---	✓	✓	---	---	✓	
	30		✓	✓	---	---	✓	---	---	✓	✓	---	---	✓	Success
	42		✓	✓	---	---	✓	---	---	✓	---	✓	---	✓	Success
	72	pass 1	✓	✓	---	---	---	---	✓						Success
		pass 2					✓	---	---	✓	✓	---	---	✓	
	77	pass 1	✓	✓	---	---	✓	---	---	✓	---	---	✓		Success
		pass 2									✓	---	---	✓	

For the failed runs in table 8.2, the execution of the abort contingency was not marked as failed since there was not an actual failure (or timeout), instead the workflow was instructed to execute the abort contingency which uses the checkpointing mechanism to resume execution from the last valid checkpoint. The failure observed in instance 10 run 6 actor 2 happened during execution of that instance (for that case, the suggested contingency was the same as the default contingency, which is why the latter got executed), once that contingency timed-out, the workflow executed the abort contingency.

The same argument from table 8.1 regarding actors 1, 4 and 6 applies to table 8.2. Since they have a very low failure probability (0.0025), we did not witness them fail in the experimental runs performed. If the number of runs were to increase (up to a 1000 or more), the granularity of this test case would be finer and we would see example where those actors might fail. However executing that many runs for all the scenarios/instances would require much larger computing resources and time than what is currently available.

Figure 8.14 shows a comparison between the average runtimes for the different instances of scenarios 2 and 3 when no failure correlation exists. The average runtime per successful run is calculated by taking the overall runtime of a 100 runs and dividing it by the number of successful runs.

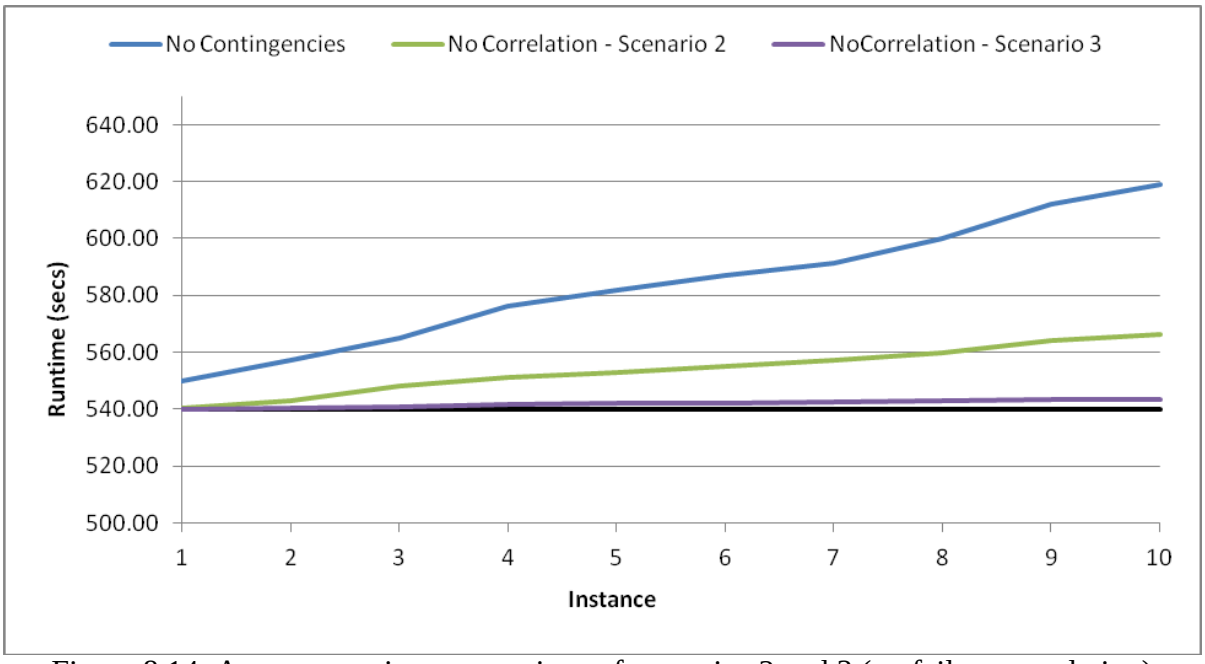

Figure 8.14: Average runtime comparison of scenarios 2 and 3 (no failure correlation)

Comparing the average runtimes of scenario 2 (green line) and scenario 3 (purple line), the runtime at instance 10 is reduced from 562 seconds to 543 seconds, which represents a drop from a 4.1% to 0.01% in runtime increase over the average run duration of a successful run when no errors occur (black line). The difference between the two scenarios would be even more significant for production workflow because of the larger number of actors, higher timeout values, failure propagation and failure correlation.

Next we compare the overhead between the two scenarios, scenario 3 uses more CPU cycles (mainly for the error-state handling layer) and more disk space due to the increased number of database entries. However since the error-state handling layer runs on a low-end machine

and since the total overhead is very minimal (less than 3%) compared to the resources used by the workflow, we will assume it is ignored.

The results observed in this scenario show that the presented fault tolerance framework significantly increase the workflow reliability, thus dramatically reducing the average workflow runtime per successful run, almost to the point where the time overhead presented by failed run is insignificant. Furthermore the overhead of the fault tolerance framework is marginal at best and can be ignored.

The simulation runs were limited to 100 runs per failure probability and correlation instance due to resources and time constraints, which does not always accurately represent the availability of actors with very low failure probability (actors 1, 4 and 6). In order for the simulations to be more sensitive to those values, we need to have a minimum of 10^3 per scenario, which is not currently feasible. However, the runs executed are a good indicator of performance and the results fall within the simulation error bounds, thus confirming the consistency of the presented model.

Chapter 9

Conclusion and future work

In this dissertation we have presented and assessed a fault tolerance framework for dataflow-based scientific workflows. We identified a number of failure scenarios that are not handled well, or at all, at the workflow level. We developed a hybrid fault tolerance approach that provides fault tolerance for workflow, middleware and hardware layers of an infrastructure collectively. This framework is composed of the following components:

1. Forward recovery mechanism: Described in chapter 5, it introduces a retry/alternative version mechanism to Kepler in the form of a contingency actor, allowing the workflow to recover on the fly from errors encountered and to resume execution.

2. Checkpointing mechanism: Described in chapter 6, it introduces a checkpointing mechanism to Kepler, which allows failed workflow to resume the execution from the last successful checkpoint, instead of re-executing the entire workflow.

3. Error-state handling layer: Described in chapter 7, it enables the workflow system to monitor the external components on which the workflow execution depends, thus allowing

the workflow to be aware of errors occurring in the external components and to address those errors when possible.

Although this framework was implemented for Kepler-based workflows, the methodology can be applied to a variety of scientific workflow management systems. The key aspect of the framework is its capability to bridge the gap between the different layers of a scientific workflow execution infrastructure by propagating sufficient information about the failures at different layers, thus allowing the workflow layer to either compensate for those failures or gracefully terminate. The efficient integration of failure handling strategies into workflow designs is a critical step for efficient and comprehensive fault tolerance. We also presented an efficient checkpointing mechanism for scientific workflows that can handle stateful actors by relying on the provenance data captured during workflow runs.

Future work includes supporting additional computational models by the checkpointing mechanism, further developing the presented framework and integrating it into the production environment in order to apply it to production level workflows such as XGC and S3D. Future work could also include extending the current framework and applying it to other scientific workflow management systems.

References

[1] Scientific Process Automation (SPA). http://sdm.lbl.gov/sdmcenter/ last referenced Jan 2011

[2] Kepler Project. http://kepler-project.org/ last referenced Jan 2011

[3] Eker J, Janneck J,, Lee E, Liu J, Liu X, Ludvig J, Neuendorffer S, Sachs S, and Xiong Y. "Taming heterogeneity – the ptolemy approach". In Proceedings of the IEEE, volume 91(1), pp 127-144, January 2003.

[4] Ilkay Altintas, Oscar Barney, Efrat Jaeger-Frank, Provenance Collection Support in the Kepler Scientific Workflow System, International Provenance and Annotation Workshop (IPAW'06), Volume 4145/2006, pp 118-132, May 2006

[5] Klasky S, Barreto R, Kahn A, Parashar M, Podhorszki N, Parker S, Silver D, Vouk, M. "Collaborative visualization spaces for petascale simulations". International Symposium on Collaborative Technologies and Systems, 2008. pp 203-211. May 2008.

[6] Taylor I. Deelman, E. Gannon, D. and Shields M. "Workflows for e-Science". ISBN: 978-1-84628-519-6, 2007

[7] Vouk M, "Software Reliability Engineering of Numerical Systems," Chapter 13, in Accuracy and Reliability in Scientific Computing, Editor: Bo Einarsson, ISBN 0-89871-584-9, pp. 205-231, SIAM, 2005

[8] Vouk M. et al. "Automation of Network-Based Scientific Workflows," in IFIP, Vol 239, "Grid-Based Problem Solving Environments, eds. Gaffney PW and Pool JCT (Boston: Springer), pp. 35-61, 2007

[9] Mouallem P., Vouk M. "Fault Tolerance in Scientific Workflows", Proc. of ETFS 2006, pp. 27-41, June 2006.

[10] Mouallem P., "Fault Tolerance and Reliability in Scientific Workflows", Master's Thesis, Dept of Computer Science, North Carolina State University, May 2005.

[11] Mouallem P., Crawl D., Altintas I., Vouk M. and Yildiz U. "Fault-Tolerance in Dataflow-based Scientific Workflow Management". SWF 2010, ISBN 978-1-4244-8199-6, pp. 336 – 343, 2010.

[12] Mouallem P., Crawl D., Altintas I., Vouk M. and Yildiz U. "A Fault-Tolerance Architecture for Kepler-based Distributed Scientific Workflows". SSDBM 2010, LNCS 6187, pp. 452-460, 2010

[13] Mouallem P., Barreto R., Klasky S., Podhorszki N. and Vouk M., “Tracking Files in the Kepler Provenance Framework”, SSDBM 2009, LNCS 5566, pp 273-282, 2009

[14] Altintas I., Chin G., Crawl D., Critchlow T., Koop D., Ligon J., Ludaescher B., Mouallem P., Nagappan M., Podhorszki N., Silva C. and Vouk M., "Provenance in Kepler-based Scientific Workflow Systems,” Poster # 41, at Microsoft eScience Workshop Friday Center, University of North Carolina, Chapell Hill, NC, October 13 - 15, 2007, pp. 82.

[15] Barreto R., Klasky S., Mouallem P., Podhorszki N. and Vouk M., “Collaboration Portal for Petascale Simulations”, CTS 2009, pp 384-393, May 2009

[16] Barreto R., Khan A., Klasky S., Mouallem P., Podhorszki N., Santos E., Silva C., and Vouk M. “Collaborative Monitoring and Analysis for Simulation Scientists”. CTS 2010, ISBN 978-1-4244-6619-1, pp. 235 – 244, 2010

[17] R. Barreto, T. Critchlow, A. Khan, S. Klasky, L. Kora, J. Ligon, P. Mouallem, M. Nagappan, N. Podhorszki, M. Vouk, "Managing and Monitoring Scientific Workflows through Dashboards," Poster # 93, at Microsoft eScience Workshop Friday Center, University of North Carolina, Chapell Hill, NC, October 13 - 15, 2007, pp. 108.

[18] M. Vouk, I. Altintas, R. Barreto, J. Blondin, Z.Cheng, T. Critchlow, A. Khan, S. Klasky, J. Ligon, B. Ludaescher, P. Mouallem, S. Parker, N. Podhorszki, A. Shoshani, C. Silva. “Automation of Network-Based Scientific Workflows”. Grid-based Problem Solving Environments, IFIP, Volume 239, pp 35-61, 2007

[19] Altintas I., Critchlow T., Khan A., Klasky S., Ligon J., Ludaescher B., Mouallem P., Nagappan M., Podhorszki N., Silva C. and Vouk M. “Introduction to Scientific Worflow Management” International Conference for High Performance Computing (SC08), Austin 2008

[20] Altintas I., Barreto R., Breimyer P., Critchlow T., Crawl D., Khan A., Koop D., Klasky S., Ligon J., Ludaescher B., Mouallem P., Nagappan M., Parker S., Podhorszki N., Silva C. and Vouk M. “Introduction to Scientific Workflow Management and the Kepler System” International Conference for High Performance Computing (SC07), Reno 2007

[21] Altintas I., Critchlow T., Khan A., Klasky S., Ligon J., Ludaescher B., Mouallem P., Parker S., Podhorszki N. and Vouk M. “Introduction to Scientific Workflow

Management and the Kepler System" International Conference for High Performance Computing (SC06), Tampa 2006

[22] Vouk M, "Software Reliability Engineering of Numerical Systems," Chapter 13, in Accuracy and Reliability in Scientific Computing, Editor: Bo Einarsson, ISBN 0-89871-584-9, SIAM, 2005, pp. 205-231

[23] NCCS at ORNL, Jaguar Supercomputer. http://www.nccs.gov/computing-resources/jaguar/ Last referenced Mar 2011

[24] Y. Zhang, M. Squillante, A. Sivasubramaniam, and R. Sahoo, "Performance implications of failures in large-scale cluster scheduling". In Proc. 10th Workshop on Job Scheduling Strategies for Parallel Processing, Vol. 3277/2005, pp. 23-38, June 2005.

[25] R. Sahoo, A. Sivasubramaniam, M. Squillante, and Y. Zhang. "Failure data analysis of a large-scale heterogeneous server environment". In Proc. of DSN'04, ISBN 0-7695-2052-9, July 2004.

[26] D. Nurmi, J. Brevik, and R. Wolski. "Modeling machine availability in enterprise and wide-area distributed computing environments". In Euro-Par 05, Volume 3648/2005, pp.612, 2005.

[27] LANL, raw data and information on failures in High performance computing systems. http://www.lanl.gov/projects/computerscience/data/ last referenced Jun 2010

[28] NERSC, raw data and information on I/O related failures, http://pdsi.nersc.gov/, last reference Jun 2010

[29] B. Schroeder, G. Gibson, "A large-scale study of failures in high-performance computing systems," International Conference on Dependable Systems and Networks (DSN'06), pp.249-258, 2006

[30] B. Schroeder, G. Gibson. "Understanding Failures in Petascale Computers". SciDAC 2007. Journal of Physics: Conference Series 78 (2007) 012022.

[31] D. Reed, "Grids, the TeraGrid and beyond," *Computer*, vol. 36, no. 1, pp. 62- 68, Jan. 2003

[32] K. Droegemeier, D. Gannon, et al. "Service-Oriented Environments for Dynamically Interacting with Mesoscale Weather," IEEE Computational Science and Engineering, vol. 7, no. 6, pp. 12-29, Dec. 2005

[33] G. Kandaswamy, A. Mandal, and D. Reed. "Fault tolerance and recovery of scientific workflows on computational grids". In CCGRID'08: Proceedings of the 2008 Eighth IEEE International Symposium on Cluster Computing and the Grid (CCGRID*)*, pages 777–782, Washington, DC, USA, 2008.

[34] O. Khalili, J. Olschanowsky, C. Snavely and H. Casanova. “Measuring the Performance and Reliability of ProductionComputational Grids,” Proceedings of the 7th IEEE/ACM International Conference on Grid Computing, ISBN:1-4244-0343-X, Sep. 2006.

[35] Geon Grid http://www.geongrid.org/ last referenced Jun 2010

[36] C. Chang, S. Ku, and H. Weitzner, “Numerical study of neoclassical plasma pedestal in a tokamak geometry”, Phys. Plasmas 11, 2649 - 2667 (2004).

[37] Center for Plasma Edge Simulation, CPES. http://www.cims.nyu.edu/cpes/. Last referenced Feb 2011

[38] Chen J. et al. “Terascale direct numerical simulations of turbulent combustion using S3D”. Computational Science & Discovery 2 015001 (31pp), Jan 2009

[39] Cummings, J. et al. “Plasma Edge Kinetic-MHD Modeling in Tokamaks Using Kepler Workflow for Code Coupling, Data Management and Visualization”, Communications in Computational Physics, 4 (3). pp. 675-702. ISSN 1815-2406

[40] Podhorszki N., Ludäscher B., Klasky S.,” Workflow Automation for Processing Plasma Fusion Simulation Data”, In: 2nd Workshop on Workflows in Support of Large-Scale Science (WORKS'07), ISBN: 978-1-59593-715-5, June 2007.

[41] Shoshani A., Rotem D, “Scientific Data Management: Challenges, Technology, and Deployment “ ISBN: 1420069802 DDC: 502.85 Edition: Hardcover; 2009-12-17

[42] Koren, I., Krishna, C.M, “Fault Tolerant Systems”. Morgan Kaufmann Publishers Inc., San Francisco 2007

[43] Frey, J. “Condor DAGMan: Handling inter-job dependencies”, http://www.bo.infn.it/calcolo/condor/dagman/ Last referenced May 2011

[44] Douglas Tain, Todd Tannenbaum, and Miron Livny, "Distributed Computing in Practice: The Condor Experience" Concurrency and Computation: Practice and Experience, Vol. 17, No. 2-4, pages 323-356, February-April, 2005.

[45] Deelman E, Singh G, Su M, Blythe J, Gil Y, Kesselman C, Mehta G, Vahi K, Berriman G, Good J, Laity A, Jacob J, and Katz D, “Pegasus: A Framework for Mapping Complex Scientific Workflows onto Distributed Systems,” Scientific Programming J., vol. 13, no. 3, pp. 219-237, 2005

[46] Kacsuk P, Sipos G. “Multi-Grid, Multi-User Workflows in the P-GRADE Grid Portal, Journal of Grid Computing, Feb 2006, pp. 1-18.

[47] Hernandez, I., Cole, M. “Reliable DAG scheduling on grids with rewinding and

migration". In Proceedings of the first international conference on Networks for grid applications. GridNets '07, ICST (2007) 3:1-3:8

[48] "The Condor Project: Job Recovery with Rescue DAG". http://www.cs.wisc.edu/condor/manual/v6.2/2_10Inter_job_Dependencies.html, last referenced Dec 2010.

[49] Y. Zhao, M. Hategan, B. Clifford, I. Foster, G. von Vonlaszewski, I. Raicu, T. Stef-Praun, and M. Wilde, "Swift: Fast, Reliable, Loosely Coupled Parallel Computation" Proc. IEEE Int'l Workshop Scientific Workflows (SWF '07), pp. 199-206, 2007.

[50] "KaraJan Execution Engine". http://www.gridworkflow.org/snips/gridworkflow/space/Karajan. last referenced Dec 2011

[51] Raicu, I., Dumitrescu, C. and Foster, I., "Dynamic Resource Provisioning in Grid Environments", TeraGrid Conference 2007.

[52] T. Fahringer, R. Prodan, R. Duan, F. Nerieri, S. Podlipnig, J. Qin, M. Siddiqui, H.-L. Truong, A. Villazon, M. Wieczorek. "ASKALON: A Grid Application Development and Computing Environment". 6th International Workshop on Grid Computing, IEEE Computer Society Press, Nov 2005

[53] Project Trident: A Scientific Workflow Workbench. Microsoft Research, 2011. http://research.microsoft.com/en-us/collaboration/tools/trident.aspx. last referenced Feb 2011

[54] K. Droegemeier, D. Gannon, D. Reed, B. Plale, J. Alameda, T. Baltzer, K. Brewster, R. Clark, B. Domenico, S. Graves, E. Joseph, D. Murray, R. Ramachandran, M. Ramamurthy, L. Ramakrishnan, J. A. Rushing, D. Weber, R. Wilhelmson, A. Wilson, M. Xue, and S. Yalda, "Service-Oriented Environments for Dynamically Interacting with Mesoscale Weather," IEEE Computational Science and Engineering, vol. 7, no. 6, pp. 12-29, Dec. 2005

[55] Feng, T., Lee, E. "Real-Time Distributed Discrete-Event Execution with Fault Tolerance". In: Real-Time and Embedded Technology and Applications Symposium, 2008. RTAS '08. IEEE. (2008) pp. 205 – 214

[56] T. Tavares, G. Teodoro, T. Kurc, R. Ferreira, D. Guedes, W. Meira Jr., U. Catalyurek, S. Hastings, S. Oster, S. Langella, J. Saltz. "An efficient and reliable scientific workflow system", in Intl. Symp. on Cluster Computing and the Grid, CCGrid, 2007.

[57] R. Ferreira, D. Guedes, L. Drummond, B. Coutinho, G. Teodoro, T. Tavares, R. Arajo, and G. Ferreira. "Anthill: A scalable run-time environment for data mining

applications". In 17th International Symposium on Computer Architecture and High Performance Computing, Rio de Janeiro, RJ, 2005.

[58] R. Tolosana, J. Banares, P. Alvarez, J. Ezpeleta, O. Rana. "An uncoordinated asynchronous checkpointing model for hierarchical scientific workflows". Journal of Computer and System Sciences, Volume 76, Issue 6, September 2010, Pages 403-415

[59] McAllister D.F. and Vouk M.A., "Software Fault-Tolerance Engineering," Chapter 14 in Handbook of Software Reliability Engineering, McGraw Hill, pp. 567-614, January 1996

[60] Randell B., "Design-Fault Tolerance," in The Evolution of Fault-Tolerant Computing, Springer-Verlag, Vienna, 1987, pp. 251-270.

[61] T. Oinn, M. Addis, J. Ferris, D. Marvin, M. Senger, R. Greenwood, T. Carver, K. Glover, M. Pocock, A. Wipat, and P. Li, "Taverna: A Tool for the Composition and Enactment of Bioinformatics Workflows," Bioinformatics, vol. 20, no. 17, pp. 3045-3054, 2004.

[62] Oinn T. et al. "Taverna: Lessons in Creating a Workflow Environment for the Life Sciences," J. Concurrency and Computation: Practice and Experience, vol. 18, no. 10, pp. 1067-1100, 2002.

[63] Taylor, M. Shields, I. Wang, and A. Harrison, "The Triana Workflow Environment: Architecture and Applications". Workflows for e-Science, pages 320-339. Springer, New York, Secaucus, NJ, USA, 2007.

[64] G. Allen and. al. "Enabling Applications on the Grid: A GridLab Overview". International Journal of High Performance Computing Applications: Special issue on Grid Computing: Infrastructure and Applications, pp. 449-466, August 2003

[65] gLite middleware for Grid Computing. http://glite.cern.ch/. Last referenced March 2011

[66] Globus Toolkit". http://www.globus.org/ last referenced Dec 2010

[67] S. Callahan, J. Freire, E. Santos, C. Scheidegger, C. Silva, and H. Vo, "VisTrails: Visualization Meets Data Management," Proc. Special Interest Group on Management of Data Conf. (SIGMOD '06), pp. 745-747, 2006.

[68] D. Laforenza, R. Lombardo, M. Scarpellini, M. Serrano, F. Silvestri, P. Faccioli. "Biological Experiments on the Grid: A Novel Workflow Management Platform", pp.

489-494, Twentieth IEEE International Symposium on Computer-Based Medical Systems (CBMS'07), 2007

[69] Dong Ruan, Shiyong Lu. “Task Exception Handling in the VIEW Scientific Workflow System”. IEEE SCC 2010, pages 637-638

[70] “BPEL: Business Process Execution Language”. http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html last referenced Dec 2009

[71] Nagappan Meiyappan and Mladen Vouk, "A Privacy Policy Model for Sharing of Provenance Information in a Query Based System", Short Paper and Poster in the Second International Provenance and Annotation Workshop (IPAW’08), Salt Lake City, UT, June 17-18, 2008.

[72] Avizienis A, “The N-Version Approach to Fault-Tolerant Systems,” IEEE Trans. Software Engineering, Vol. SE- 1 1, No. 12, pp. 1,491-1,501, Dec. 1985.

[73] Yibei L, Jie M. and Xiaola L. “A Variational Calculus Approach to Optimal Checkpoint Placement’. IEEE Trans. Computers 50(7), pp. 699-708, 2001.

[74] Lee, E., Matsikoudis, E. “The semantics of dataflow with firing”. From Semantics to Computer Science: Essays in memory of Gilles Kahn. Cambridge University Press, Cambridge (2008)

[75] Lee, E.A., Messerschmitt, D.G. “Static scheduling of synchronous dataflow programs for digital signal processing”. IEEE Trans. Comput. 36 pp. 24-35 , 1987.

[76] Zhou, G. “Dynamic dataflow modeling in Ptolemy II”. PhD thesis, University of California (2004)

[77] Barr M. “Introduction to Watchdog Timers”. http://www.embedded.com/story/OEG20010920S0064, last referenced Jan 2011

[78] NCSU Virtual Computing Laboratory (VCL), http://vcl.ncsu.edu, last referenced April 2011

[79] Voas J., "Fault Injection for the Masses," *Computer*, vol. 30, no. 12, pp. 129-130, Dec. 1997

[80] Xmgrace plotting tool, http://plasma-gate.weizmann.ac.il/Grace/, last referenced April 2011

[81] AVS Converter, http://www.avs.com/, last referenced April 2011

[82] Trivedi K. “Probability and Statistics with Reliability, Queuing, and Computer Science Applications”, 2nd Edition, John Wiley and Sons, New York, 2001, ISBN number 0-471-33341-7.

Appendix

Appendix A

XGC Workflow Production Run Log Extracts

1. Successful Run (duration 2h35)

File: /logs/shotpf89/log/run001/workflow.log

Note that the log file has 8373 entries, therefore we will only show a small extract below

```
Log file created on Mar 21 2011 00:04:34.609
Log date format: MMM dd yyyy HH:mm:ss.SSS
-------------------------------------------------
Mar 21 2011 00:04:34.609: Create Directory: local:/tmp/work-
local/shku/workflow/xgc1/shotpf89/bp has been created.
Mar 21 2011 00:04:34.614: Create Directory: local:/tmp/work-
local/shku/workflow/xgc1/shotpf89/bp has been created.
Mar 21 2011 00:04:34.628: Create Directory: local:/tmp/work-
local/shku/workflow/xgc1/shotpf89/hdf5 has been created.
Mar 21 2011 00:04:34.632: Create Directory: local:/tmp/work-
local/shku/workflow/xgc1/shotpf89/hdf5 has been created.
Mar 21 2011 00:04:34.636: Create Directory: local:/tmp/work-
local/shku/workflow/xgc1/shotpf89/image has been created.
Mar 21 2011 00:04:34.641: Create Directory: local:/tmp/work-
local/shku/workflow/xgc1/shotpf89/avs has been created.
Mar 21 2011 00:04:34.647: Create Directory: local:/tmp/work-
local/shku/workflow/xgc1/shotpf89/txt/run001 has been created.
Mar 21 2011 00:04:34.652: Create Directory: local:/tmp/work-
local/shku/workflow/xgc1/shotpf89/image/global has been created.
Mar 21 2011 00:04:57.843: Split_1D:  start: cd /tmp/work/shku/pf89_restart_temp_pf88;
/sw/xt5/adios/1.2.1/cnl2.2_pgi10.4/bin/bpsplit -v -v -r xgc.flowdiag.bp.splitidx xgc.flowdiag.bp
xgc.flowdiag.spl >> xgc.flowdiag.bp.splitlog
Mar 21 2011 00:04:58.451: StartAVS3D: AVS job submission skipped.
Mar 21 2011 00:05:00.354: Split_1D:  succ: cd /tmp/work/shku/pf89_restart_temp_pf88;
/sw/xt5/adios/1.2.1/cnl2.2_pgi10.4/bin/bpsplit -v -v -r xgc.flowdiag.bp.splitidx xgc.flowdiag.bp
xgc.flowdiag.spl >> xgc.flowdiag.bp.splitlog
Mar 21 2011 00:05:02.874: Transfer_1D:  start: /sw/sith/wf-xgc-
monitor/3.1/centos5.5_binary/bin/retry.sh 3 60 scp -r
/tmp/work/shku/pf89_restart_temp_pf88/xgc.flowdiag.spl sith:/tmp/work-
local/shku/workflow/xgc1/shotpf89/bp/xgc.flowdiag.spl
Mar 21 2011 00:05:02.877: Split_1D:  start: cd /tmp/work/shku/pf89_restart_temp_pf88;
/sw/xt5/adios/1.2.1/cnl2.2_pgi10.4/bin/bpsplit -v -v -r xgc.fluxdiag.bp.splitidx xgc.fluxdiag.bp
xgc.fluxdiag.spl >> xgc.fluxdiag.bp.splitlog
Mar 21 2011 00:05:05.942: Split_1D:  succ: cd /tmp/work/shku/pf89_restart_temp_pf88;
/sw/xt5/adios/1.2.1/cnl2.2_pgi10.4/bin/bpsplit -v -v -r xgc.fluxdiag.bp.splitidx xgc.fluxdiag.bp
xgc.fluxdiag.spl >> xgc.fluxdiag.bp.splitlog
```

```
Mar 21 2011 00:05:07.456: Split_1D:  start: cd /tmp/work/shku/pf89_restart_temp_pf88;
/sw/xt5/adios/1.2.1/cnl2.2_pgi10.4/bin/bpsplit -v -v -r xgc.oneddiag.bp.splitidx xgc.oneddiag.bp
xgc.oneddiag.spl >> xgc.oneddiag.bp.splitlog
Mar 21 2011 00:05:08.398: Transfer_1D:  succ: /sw/sith/wf-xgc-
monitor/3.1/centos5.5_binary/bin/retry.sh 3 60 scp -r
/tmp/work/shku/pf89_restart_temp_pf88/xgc.flowdiag.spl sith:/tmp/work-
local/shku/workflow/xgc1/shotpf89/bp/xgc.flowdiag.spl
Mar 21 2011 00:05:08.408: Transfer_1D:  start: /sw/sith/wf-xgc-
monitor/3.1/centos5.5_binary/bin/retry.sh 3 60 scp -r
/tmp/work/shku/pf89_restart_temp_pf88/xgc.fluxdiag.spl sith:/tmp/work-
local/shku/workflow/xgc1/shotpf89/bp/xgc.fluxdiag.spl
Mar 21 2011 00:05:08.409: Append_1D:  start: cd /tmp/work-
local/shku/workflow/xgc1/shotpf89/bp; bpappend -v -v xgc.flowdiag.spl xgc.flowdiag.bp >>
xgc.flowdiag.spl.appendlog
Mar 21 2011 00:05:10.170: Append_1D:  succ: cd /tmp/work-
local/shku/workflow/xgc1/shotpf89/bp; bpappend -v -v xgc.flowdiag.spl xgc.flowdiag.bp >>
xgc.flowdiag.spl.appendlog
Mar 21 2011 00:05:10.177: Graph_1D:  start: cd /tmp/work-
local/shku/workflow/xgc1/shotpf89/log/run001; /sw/sith/wf-xgc-
monitor/3.1/centos5.5_binary/bin/runscriptasjob.sh /sw/sith/wf-xgc-
monitor/3.1/centos5.5_binary/bin/graph-1d-xgc.sh graph1djobs.log -i 2 -c 2150 /tmp/work-
local/shku/workflow/xgc1/shotpf89/bp/xgc.flowdiag.bp /tmp/work-
local/shku/workflow/xgc1/shotpf89/image
Mar 21 2011 00:05:10.471: Split_1D:  succ: cd /tmp/work/shku/pf89_restart_temp_pf88;
/sw/xt5/adios/1.2.1/cnl2.2_pgi10.4/bin/bpsplit -v -v -r xgc.oneddiag.bp.splitidx xgc.oneddiag.bp
xgc.oneddiag.spl >> xgc.oneddiag.bp.splitlog
```

...

```
Mar 21 2011 02:11:22.324: Graph_2D_Global_i2:  succ: cd /tmp/work-
local/shku/workflow/xgc1/shotpf89/log/run001; /sw/sith/wf-xgc-
monitor/3.1/centos5.5_binary/bin/runscriptasjob.sh /sw/sith/wf-xgc-
monitor/3.1/centos5.5_binary/bin/graph-2d-xgc.sh graph2djobs.log -g /tmp/work-
local/shku/workflow/xgc1/shotpf89/hdf5/xgc.fieldp.0501.h5 /tmp/work-
local/shku/workflow/xgc1/shotpf89/image
Mar 21 2011 02:11:23.684: Graph_2D_Global_i0:  succ: cd /tmp/work-
local/shku/workflow/xgc1/shotpf89/log/run001; /sw/sith/wf-xgc-
monitor/3.1/centos5.5_binary/bin/runscriptasjob.sh /sw/sith/wf-xgc-
monitor/3.1/centos5.5_binary/bin/graph-2d-xgc.sh graph2djobs.log -g /tmp/work-
local/shku/workflow/xgc1/shotpf89/hdf5/xgc.fieldp.0493.h5 /tmp/work-
local/shku/workflow/xgc1/shotpf89/image
Mar 21 2011 02:11:27.150: Graph_2D_Global_i6:  succ: cd /tmp/work-
local/shku/workflow/xgc1/shotpf89/log/run001; /sw/sith/wf-xgc-
monitor/3.1/centos5.5_binary/bin/runscriptasjob.sh /sw/sith/wf-xgc-
monitor/3.1/centos5.5_binary/bin/graph-2d-xgc.sh graph2djobs.log -g /tmp/work-
local/shku/workflow/xgc1/shotpf89/hdf5/xgc.fieldp.0456.h5 /tmp/work-
local/shku/workflow/xgc1/shotpf89/image
Mar 21 2011 02:11:33.997: Graph_2D_Global_i1:  succ: cd /tmp/work-
local/shku/workflow/xgc1/shotpf89/log/run001; /sw/sith/wf-xgc-
monitor/3.1/centos5.5_binary/bin/runscriptasjob.sh /sw/sith/wf-xgc-
monitor/3.1/centos5.5_binary/bin/graph-2d-xgc.sh graph2djobs.log -g /tmp/work-
```

local/shku/workflow/xgc1/shotpf89/hdf5/xgc.fieldp.0411.h5 /tmp/work-
local/shku/workflow/xgc1/shotpf89/image
Mar 21 2011 02:11:34.000: Graph_2D_Global_i1: start: cd /tmp/work-
local/shku/workflow/xgc1/shotpf89/log/run001; /sw/sith/wf-xgc-
monitor/3.1/centos5.5_binary/bin/runscriptasjob.sh /sw/sith/wf-xgc-
monitor/3.1/centos5.5_binary/bin/graph-2d-xgc.sh graph2djobs.log -g /tmp/work-
local/shku/workflow/xgc1/shotpf89/hdf5/xgc.fieldp.0498.h5 /tmp/work-
local/shku/workflow/xgc1/shotpf89/image
Mar 21 2011 02:11:34.437: Graph_2D_Global_i3: succ: cd /tmp/work-
local/shku/workflow/xgc1/shotpf89/log/run001; /sw/sith/wf-xgc-
monitor/3.1/centos5.5_binary/bin/runscriptasjob.sh /sw/sith/wf-xgc-
monitor/3.1/centos5.5_binary/bin/graph-2d-xgc.sh graph2djobs.log -g /tmp/work-
local/shku/workflow/xgc1/shotpf89/hdf5/xgc.fieldp.0415.h5 /tmp/work-
local/shku/workflow/xgc1/shotpf89/image
Mar 21 2011 02:11:34.440: Graph_2D_Global_i3: start: cd /tmp/work-
local/shku/workflow/xgc1/shotpf89/log/run001; /sw/sith/wf-xgc-
monitor/3.1/centos5.5_binary/bin/runscriptasjob.sh /sw/sith/wf-xgc-
monitor/3.1/centos5.5_binary/bin/graph-2d-xgc.sh graph2djobs.log -g /tmp/work-
local/shku/workflow/xgc1/shotpf89/hdf5/xgc.fieldp.0511.h5 /tmp/work-
local/shku/workflow/xgc1/shotpf89/image
Mar 21 2011 02:11:54.380: Graph_2D_Global_i1: succ: cd /tmp/work-
local/shku/workflow/xgc1/shotpf89/log/run001; /sw/sith/wf-xgc-
monitor/3.1/centos5.5_binary/bin/runscriptasjob.sh /sw/sith/wf-xgc-
monitor/3.1/centos5.5_binary/bin/graph-2d-xgc.sh graph2djobs.log -g /tmp/work-
local/shku/workflow/xgc1/shotpf89/hdf5/xgc.fieldp.0498.h5 /tmp/work-
local/shku/workflow/xgc1/shotpf89/image
Mar 21 2011 02:11:54.824: Graph_2D_Global_i3: succ: cd /tmp/work-
local/shku/workflow/xgc1/shotpf89/log/run001; /sw/sith/wf-xgc-
monitor/3.1/centos5.5_binary/bin/runscriptasjob.sh /sw/sith/wf-xgc-
monitor/3.1/centos5.5_binary/bin/graph-2d-xgc.sh graph2djobs.log -g /tmp/work-
local/shku/workflow/xgc1/shotpf89/hdf5/xgc.fieldp.0511.h5 /tmp/work-
local/shku/workflow/xgc1/shotpf89/image
Mar 21 2011 02:11:54.865: GlobalMovies: start: cd /tmp/work-
local/shku/workflow/xgc1/shotpf89/log/run001; /sw/sith/wf-xgc-
monitor/3.1/centos5.5_binary/bin/runscriptasjob.sh /sw/sith/wf-xgc-
monitor/3.1/centos5.5_binary/bin/movie.sh moviejobs.log -d /tmp/work-
local/shku/workflow/xgc1/shotpf89/image -i global -l /tmp/work-
local/shku/workflow/xgc1/shotpf89/log/run001/movie_g.log
Mar 21 2011 02:11:54.894: StopAVS3D: AVS job removal is skipped
Mar 21 2011 02:36:23.648: GlobalMovies: succ: cd /tmp/work-
local/shku/workflow/xgc1/shotpf89/log/run001; /sw/sith/wf-xgc-
monitor/3.1/centos5.5_binary/bin/runscriptasjob.sh /sw/sith/wf-xgc-
monitor/3.1/centos5.5_binary/bin/movie.sh moviejobs.log -d /tmp/work-
local/shku/workflow/xgc1/shotpf89/image -i global -l /tmp/work-
local/shku/workflow/xgc1/shotpf89/log/run001/movie_g.log
Mar 21 2011 02:36:23.651: ArchiveImages: start: cd /tmp/work-
local/shku/workflow/xgc1/shotpf89/image; hsi -q mkdir -p xgc1/pf89; htar -c -f
xgc1/pf89/images.tar .
Mar 21 2011 02:39:26.673: ArchiveImages: succ: cd /tmp/work-
local/shku/workflow/xgc1/shotpf89/image; hsi -q mkdir -p xgc1/pf89; htar -c -f
xgc1/pf89/images.tar .
---

2. Failed Run (duration 0h53)

File: /logs/shotpf88_2/log/run001/workflow.log

Note that the log file has 4188 entries, therefore we will only show a small extract below

```
Log file created on Mar 02 2011 14:57:26.395
Log date format: MMM dd yyyy HH:mm:ss.SSS
-------------------------------------------
Mar 02 2011 14:57:26.396: Create Directory: local:/tmp/work-
local/shku/workflow/xgc1/shotpf88_2/bp has been created.
Mar 02 2011 14:57:26.400: Create Directory: local:/tmp/work-
local/shku/workflow/xgc1/shotpf88_2/bp has been created.
Mar 02 2011 14:57:26.418: Create Directory: local:/tmp/work-
local/shku/workflow/xgc1/shotpf88_2/hdf5 has been created.
Mar 02 2011 14:57:26.422: Create Directory: local:/tmp/work-
local/shku/workflow/xgc1/shotpf88_2/hdf5 has been created.
Mar 02 2011 14:57:26.427: Create Directory: local:/tmp/work-
local/shku/workflow/xgc1/shotpf88_2/image has been created.
Mar 02 2011 14:57:26.432: Create Directory: local:/tmp/work-
local/shku/workflow/xgc1/shotpf88_2/avs has been created.
Mar 02 2011 14:57:26.439: Create Directory: local:/tmp/work-
local/shku/workflow/xgc1/shotpf88_2/txt/run001 has been created.
Mar 02 2011 14:57:26.444: Create Directory: local:/tmp/work-
local/shku/workflow/xgc1/shotpf88_2/image/global has been created.
Mar 02 2011 14:57:29.592: Split_1D:  start: cd /tmp/work/shku/pf88_wo_pade_pf45;
/sw/xt5/adios/1.2.1/cnl2.2_pgi10.4/bin/bpsplit -v -v -r xgc.flowdiag.bp.splitidx xgc.flowdiag.bp
xgc.flowdiag.spl >> xgc.flowdiag.bp.splitlog
Mar 02 2011 14:57:30.306: StartAVS3D: AVS job submission skipped.
Mar 02 2011 14:57:32.103: Split_1D:  succ: cd /tmp/work/shku/pf88_wo_pade_pf45;
/sw/xt5/adios/1.2.1/cnl2.2_pgi10.4/bin/bpsplit -v -v -r xgc.flowdiag.bp.splitidx xgc.flowdiag.bp
xgc.flowdiag.spl >> xgc.flowdiag.bp.splitlog
Mar 02 2011 14:57:33.616: Transfer_1D:  start: /sw/sith/wf-xgc-
monitor/3.1/centos5.5_binary/bin/retry.sh 3 60 scp -r
/tmp/work/shku/pf88_wo_pade_pf45/xgc.flowdiag.spl sith:/tmp/work-
local/shku/workflow/xgc1/shotpf88_2/bp/xgc.flowdiag.spl
Mar 02 2011 14:57:33.619: Split_1D:  start: cd /tmp/work/shku/pf88_wo_pade_pf45;
/sw/xt5/adios/1.2.1/cnl2.2_pgi10.4/bin/bpsplit -v -v -r xgc.fluxdiag.bp.splitidx xgc.fluxdiag.bp
xgc.fluxdiag.spl >> xgc.fluxdiag.bp.splitlog
Mar 02 2011 14:57:36.182: Split_1D:  succ: cd /tmp/work/shku/pf88_wo_pade_pf45;
/sw/xt5/adios/1.2.1/cnl2.2_pgi10.4/bin/bpsplit -v -v -r xgc.fluxdiag.bp.splitidx xgc.fluxdiag.bp
xgc.fluxdiag.spl >> xgc.fluxdiag.bp.splitlog
Mar 02 2011 14:57:37.134: Transfer_1D:  succ: /sw/sith/wf-xgc-
monitor/3.1/centos5.5_binary/bin/retry.sh 3 60 scp -r
/tmp/work/shku/pf88_wo_pade_pf45/xgc.flowdiag.spl sith:/tmp/work-
local/shku/workflow/xgc1/shotpf88_2/bp/xgc.flowdiag.spl
Mar 02 2011 14:57:37.142: Append_1D:  start: cd /tmp/work-
local/shku/workflow/xgc1/shotpf88_2/bp; bpappend -v -v xgc.flowdiag.spl xgc.flowdiag.bp >>
xgc.flowdiag.spl.appendlog
```

```
44 Mar 02 2011 14:57:37.277: Append_1D:  fail: cd /tmp/work-
45 local/shku/workflow/xgc1/shotpf88_2/bp; bpappend -v -v xgc.flowdiag.spl xgc.flowdiag.bp >>
46 xgc.flowdiag.spl.appendlog
47 Mar 02 2011 14:57:37.694: Transfer_1D:  start: /sw/sith/wf-xgc-
48 monitor/3.1/centos5.5_binary/bin/retry.sh 3 60 scp -r
49 /tmp/work/shku/pf88_wo_pade_pf45/xgc.fluxdiag.spl sith:/tmp/work-
50 local/shku/workflow/xgc1/shotpf88_2/bp/xgc.fluxdiag.spl
51 Mar 02 2011 14:57:37.697: Split_1D:  start: cd /tmp/work/shku/pf88_wo_pade_pf45;
52 /sw/xt5/adios/1.2.1/cnl2.2_pgi10.4/bin/bpsplit -v -v -r xgc.oneddiag.bp.splitidx xgc.oneddiag.bp
53 xgc.oneddiag.spl >> xgc.oneddiag.bp.splitlog
54 Mar 02 2011 14:57:40.759: Split_1D:  succ: cd /tmp/work/shku/pf88_wo_pade_pf45;
55 /sw/xt5/adios/1.2.1/cnl2.2_pgi10.4/bin/bpsplit -v -v -r xgc.oneddiag.bp.splitidx xgc.oneddiag.bp
56 xgc.oneddiag.spl >> xgc.oneddiag.bp.splitlog
57 Mar 02 2011 14:57:41.242: Transfer_1D:  succ: /sw/sith/wf-xgc-
58 monitor/3.1/centos5.5_binary/bin/retry.sh 3 60 scp -r
59 /tmp/work/shku/pf88_wo_pade_pf45/xgc.fluxdiag.spl sith:/tmp/work-
60 local/shku/workflow/xgc1/shotpf88_2/bp/xgc.fluxdiag.spl
61 Mar 02 2011 14:57:41.252: Append_1D:  start: cd /tmp/work-
62 local/shku/workflow/xgc1/shotpf88_2/bp; bpappend -v -v xgc.fluxdiag.spl xgc.fluxdiag.bp >>
63 xgc.fluxdiag.spl.appendlog
64 Mar 02 2011 14:57:41.389: Append_1D:  fail: cd /tmp/work-
65 local/shku/workflow/xgc1/shotpf88_2/bp; bpappend -v -v xgc.fluxdiag.spl xgc.fluxdiag.bp >>
66 xgc.fluxdiag.spl.appendlog

   ....

4172 Mar 02 2011 15:39:44.628: RegisterTxt:  start: /sw/sith/wf-xgc-
4173 monitor/3.1/centos5.5_binary/bin/dashboardxml.py /tmp/work-
4174 local/shku/workflow/xgc1/shotpf88_2/dashboard.xml txt/run001 text xgcs; /sw/sith/wf-xgc-
4175 monitor/3.1/centos5.5_binary/bin/register-variables -c 2145 -p /tmp/work-
4176 local/shku/workflow/xgc1/shotpf88_2/txt/run001 -t text -s -1 xgcs
4177 Mar 02 2011 15:39:44.763: RegisterTxt:  succ: /sw/sith/wf-xgc-
4178 monitor/3.1/centos5.5_binary/bin/dashboardxml.py /tmp/work-
4179 local/shku/workflow/xgc1/shotpf88_2/dashboard.xml txt/run001 text xgcs; /sw/sith/wf-xgc-
4180 monitor/3.1/centos5.5_binary/bin/register-variables -c 2145 -p /tmp/work-
4181 local/shku/workflow/xgc1/shotpf88_2/txt/run001 -t text -s -1 xgcs
4182 Mar 02 2011 15:39:44.768: ArchiveTxt:  start: cd /tmp/work-
4183 local/shku/workflow/xgc1/shotpf88_2/txt/run001; hsi -q mkdir -p xgc1/pf88_2; htar -c -f
4184 xgc1/pf88_2/txt.run001.tar .
4185 Mar 02 2011 15:39:49.597: ArchiveTxt:  succ: cd /tmp/work-
4186 local/shku/workflow/xgc1/shotpf88_2/txt/run001; hsi -q mkdir -p xgc1/pf88_2; htar -c -f
4187 xgc1/pf88_2/txt.run001.tar .
4188 Mar 02 2011 15:39:52.504: TransferExtraBP:  succ: /sw/sith/wf-xgc-
4189 monitor/3.1/centos5.5_binary/bin/retry.sh 3 60 scp -r
4190 /tmp/work/shku/pf88_wo_pade_pf45/xgc.rect.bp sith:/tmp/work-
4191 local/shku/workflow/xgc1/shotpf88_2/bp/xgc.rect.bp
4192 Mar 02 2011 15:39:52.518: Archive_1D:  start: cd /tmp/work-
4193 local/shku/workflow/xgc1/shotpf88_2/bp; hsi -q mkdir -p xgc1/pf88_2; htar -c -f
4194 xgc1/pf88_2/data1D.tar xgc.*diag.bp xgc.rect.bp xgc.bfield.bp xgc.poisson.bp
4195 xgc.edensity0.bp
4196 Mar 02 2011 15:39:58.335: MakeMovies:  succ: cd /tmp/work-
4197 local/shku/workflow/xgc1/shotpf88_2/log/run001; /sw/sith/wf-xgc-
```

monitor/3.1/centos5.5_binary/bin/runscriptasjob.sh /sw/sith/wf-xgc-
monitor/3.1/centos5.5_binary/bin/movie.sh moviejobs.log -d /tmp/work-
local/shku/workflow/xgc1/shotpf88_2 -l /tmp/work-
local/shku/workflow/xgc1/shotpf88_2/log/run001/movie.log -m 2145
Mar 02 2011 15:39:58.339: GlobalMinmax: start: cd /tmp/work-
local/shku/workflow/xgc1/shotpf88_2/image; /sw/sith/wf-xgc-
monitor/3.1/centos5.5_binary/bin/xgc-globalminmax.py minmax2D.dat | tee -a gminmax.log
Mar 02 2011 15:39:58.476: GlobalMinmax: succ: cd /tmp/work-
local/shku/workflow/xgc1/shotpf88_2/image; /sw/sith/wf-xgc-
monitor/3.1/centos5.5_binary/bin/xgc-globalminmax.py minmax2D.dat | tee -a gminmax.log
Mar 02 2011 15:39:58.521: GlobalMovies: start: cd /tmp/work-
local/shku/workflow/xgc1/shotpf88_2/log/run001; /sw/sith/wf-xgc-
monitor/3.1/centos5.5_binary/bin/runscriptasjob.sh /sw/sith/wf-xgc-
monitor/3.1/centos5.5_binary/bin/movie.sh moviejobs.log -d /tmp/work-
local/shku/workflow/xgc1/shotpf88_2/image -i global -l /tmp/work-
local/shku/workflow/xgc1/shotpf88_2/log/run001/movie_g.log
Mar 02 2011 15:39:58.541: StopAVS3D: AVS job removal is skipped
Mar 02 2011 15:39:58.574: Archive_1D: fail: cd /tmp/work-
local/shku/workflow/xgc1/shotpf88_2/bp; hsi -q mkdir -p xgc1/pf88_2; htar -c -f
xgc1/pf88_2/data1D.tar xgc.*diag.bp xgc.rect.bp xgc.bfield.bp xgc.poisson.bp
xgc.edensity0.bp
Mar 02 2011 15:40:03.761: GlobalMovies: succ: cd /tmp/work-
local/shku/workflow/xgc1/shotpf88_2/log/run001; /sw/sith/wf-xgc-
monitor/3.1/centos5.5_binary/bin/runscriptasjob.sh /sw/sith/wf-xgc-
monitor/3.1/centos5.5_binary/bin/movie.sh moviejobs.log -d /tmp/work-
local/shku/workflow/xgc1/shotpf88_2/image -i global -l /tmp/work-
local/shku/workflow/xgc1/shotpf88_2/log/run001/movie_g.log
Mar 02 2011 15:40:03.764: ArchiveImages: start: cd /tmp/work-
local/shku/workflow/xgc1/shotpf88_2/image; hsi -q mkdir -p xgc1/pf88_2; htar -c -f
xgc1/pf88_2/images.tar .
Mar 02 2011 15:40:08.501: ArchiveImages: succ: cd /tmp/work-
local/shku/workflow/xgc1/shotpf88_2/image; hsi -q mkdir -p xgc1/pf88_2; htar -c -f
xgc1/pf88_2/images.tar

File: /logs/shotpf88_2/log/run001/workflow.err

```
1  Log file created on Mar 02 2011 14:57:37.276
2  Log date format: MMM dd yyyy HH:mm:ss.SSS
3  ----------------------------------------------
4  Mar 02 2011 14:57:37.277: Append_1D: ERROR at command: cd /tmp/work-
5  local/shku/workflow/xgc1/shotpf88_2/bp; bpappend -v -v xgc.flowdiag.spl xgc.flowdiag.bp >>
6  xgc.flowdiag.spl.appendlog
7  Exit code = 127
8  Stderr: /bin/sh: bpappend: command not found
9
10 Mar 02 2011 14:57:41.389: Append_1D: ERROR at command: cd /tmp/work-
11 local/shku/workflow/xgc1/shotpf88_2/bp; bpappend -v -v xgc.fluxdiag.spl xgc.fluxdiag.bp >>
12 xgc.fluxdiag.spl.appendlog
13 Exit code = 127
14 Stderr: /bin/sh: bpappend: command not found

   …

4172 Mar 02 2011 15:39:58.574: Archive_1D: ERROR at command: cd /tmp/work-
4173 local/shku/workflow/xgc1/shotpf88_2/bp; hsi -q mkdir -p xgc1/pf88_2; htar -c -f
4174 xgc1/pf88_2/data1D.tar xgc.*diag.bp xgc.rect.bp xgc.bfield.bp xgc.poisson.bp
4175 xgc.edensity0.bp
4176 Exit code = 72
4177 Stderr: ERROR: /opt/public/bin/htar_l: error -1 stat-ing "xgc.poisson.bp"
4178
4179 ----------------------------------------------
4180
4181 Log file created on Mar 21 2011 00:04:34.609
```

File: /logs/shotpf88_2/log/run001/workflow.log.job

```
1  Log file created on Mar 02 2011 14:57:41.084
2  Log date format: MMM dd yyyy HH:mm:ss.SSS
3  ----------------------------------------------
4  Mar 02 2011 14:57:41.085: JobWatcher:  Result for qstat -a pf88_2:
5
6  stderr:
7  qstat: Unknown queue destination pf88_2
8
9  exit code: 170
10 ----------------------------------------------
```