2010

Dynamic fault tolerant grid workflow in the water threat management project

Young Suk Moon

Master's Thesis

Dynamic Fault Tolerant Grid Workflow in the Water Threat Management Project

Young Suk Moon

Department of Computer Science, Rochester Institute of Technology

Email: ysm906@gmail.com

Chair: Dr. Hans-Peter Bischof Date

Reader: Dr. Gregor von Laszewski Date

Observer: Dr. Minseok Kwon Date

Acknowledgment

I thank Dr. Gregor von Laszewski for supervising my thesis work with valuable discussion. I thank Dr. Lizhe Wang who also advised me for the improvement of my work. Additionally, I appreciate Sreerama Sreepathi from North Carolina State University who helped us to understand the Water Threat Management application.

I also thank the members of the group including Leor Dilmanian, Andrew Younge, Paresh Khatri, Fugang Wang, and Sanket Patel who have contributed to the work done as part of the Cyberaide team. It would have been very hard to progress without the help from those contributors.

Work conducted by Dr. Gregor von Laszewski and Dr. Lizhe Wang is supported (in part) by NSF CMMI 0540076 and NSF SDCI NMI 0721656.

This research used in part resources of the TeraGrid. The numbers for the TeraGrid acounts are TG-MIP050000T and TG-MIP050001.

The initial version of the Cyberaide Shell was conceptually introduced and practically demonstrated by Dr. Gregor von Laszewski at Argonne National Laboratory. At the time the project was known under the name CoG Shell (see the Appendix for more details).

Contents

List of Figures

List of Tables

Abstract

Achieving fault tolerance is an inevitable problem in distributed systems, with it becoming more challenging in decentralized, heterogeneous, and dynamic-environment systems such as a Grid. When deploying applications requires time-criticality, how to allocate resources for jobs in a fault-tolerant manner is an important issue for the delivery of the services.

The *Water Threat Management* project is a research to find solutions for the contamination incidents problems in urban water distribution systems, and it involves the development of the cyberinfrastructure in a Grid environment. To handle such urgent events properly, the deployment of the system demands real-time processing without the failure.

Our approach of integrating a fault-tolerant framework into a Water Threat Management system provides fault tolerance at the "queuing stage" rather than the "job-execution stage" by scheduling jobs in fault-tolerant ways. This includes the development of the batch queuing system in the *Cyberaide Shell* project. In addition, we present a dynamic workflow in the Water Threat Management system that can reduce the queue wait time in the changing environment.

1 Introduction

As a solution to manage contamination incidents in urban Water Distribution Systems (WDSs), research on cyberinfrastructure has been developed for the threat management - Water Threat Management system [1] [2]. The processing of the Water Threat Management system mainly consists of two stages. During the first stage, the system searches the locations of the contaminant sources. It then simulates the water flow and the quality within the pipe networks of WDSs at the next stage. This scientific application has been developed for execution in the powerful computing environment due to the need for its massive processing.

However, since fault tolerance is not provided in the system, the threat management may fail to handle the incidents properly in WDSs. The purpose of this thesis work is to design a fault-tolerant framework and integrate it into the Water Threat Management system [2] for its deployment on the TeraGrid [3].

We introduce a fault-tolerant queue that has been developed as an interface of Cyberaide Shell [4]. Fault tolerance is managed with the job scheduling in the queuing system by detecting the failure and re-scheduling the jobs. It integrates the TeraGrid [3] sites for the user to access the resources with simple Command Line Interface (CLI). The queue schedules jobs to run on the site resources and also handles job failure caused by the site outage to avoid the unavailability of the local scheduling and running jobs on the target site within the system.

We also describe a workflow of the threat management system that dynamically adapts to the changing Grid environment. This includes minimizing the queue wait time by scheduling jobs on multiple sites. It can be accomplished by determining job dependency after the job submissions. Also, the workflow is designed to be compatible with the fault-tolerant queuing system.

In the remainder of this section, for the background of this thesis topic, Grid computing is briefly described, and the Water Threat Management system [1] [2] is explained in addition to the Cyberaide Shell [4] project.

1.1 Grid Computing

Grid computing has emerged for the necessity of huge computational power and large amount of data processing for scientific and business communities [5]. It is an integration of hardware, software, and the organizations using the facilitated computer resources in a distributed environment [5].

1.1.1 Grid Definition

There is no clear definition of a Grid as the term is described in a number of papers [6] [5] [7]. Although the definition in [7] is lack of the idea of sharing, the definition of a Grid can be summarized as the use of coordinated and shared software and hardware involving the collaborative work of the users who form the Virtual Organizations (VOs) from those literatures [6] [5] [7].

1.1.2 Grid Description

To inspect more of a Grid system in addition to the definition, the three different viewpoints would help in understanding a Grid system. The perspectives include 1) who the users are; 2) the composition of the components; and 3) how they are utilized. The users of a Grid system are members of Virtual Organizations (VOs) [6]. VOs are composed of individuals or institutions forming groups, with each having the same goal in solving problems. A VO may consist of application service providers and scientists who need high-performance computer systems for their research activities. Also, an individual may be a member of multiple VOs.

From a physical viewpoint, a Grid system can be composed of supercomputers or clusters belonging to institutions in a networked environment with high bandwidth [8]. It includes local resource managers such as PBS [9] for batch job scheduling due to the fact of sharing resources among the different users. In addition, the resources may be heterogeneous in terms of hardware and software. The computers may have different architectures, operating systems, and software. Further, the overall architecture of the system in one site may also be different from another.

To facilitate the use of such a system for VOs, a Grid system coordinates the use of the resources with security policies [6]. This necessitates a Grid middleware for coordinated access to the resources in a secure manner for the heterogeneous environments.

1.1.3 Grid Middleware

A Grid provides standard protocols and interfaces for the coordination of the resources with user authentication and authorization [10]. Since the resources are shared among the groups of users, a Grid system requires security mechanisms and policies not only for individuals but also for the organizations [10]. A Grid middleware supports those requirements in addition to the composition of the heterogeneous resources belonging to different sites [10]. A widely used middleware is the Globus Toolkit [11] which is also used for the development of the fault-tolerant framework as part of this thesis work.

The main functionalities of the toolkit are secure access to the resources, job and resource management, resource discovery, information services, and data management [11]. They can be provided as Web Services [12] which can enable the development of a cyberinfrastructure [13]. We mainly facilitate the Grid Resource Allocation and Management (GRAM) service [11] of the toolkit for the failure and job management in our framework.

1.1.4 Grid Security

Security is also an integral part of a Grid. In a Grid system, several frameworks such as X.509 proxy certificates [14], MyProxy [15], and Grid Security Infrastructure (GSI) [16] based on Public Key Interface (PKI) are utilized for a number of purposes including authentication, authorization, and delegation [17]. Notably, the Globus Toolkit [11] implements GSI for its security framework, and GSI uses X.509 certificates [16] [17].

The main functions of GSI are authorization, authentication, and delegation [16]. Authorization is accomplished with the use of Security Association Markup Language (SAML) [18] and grid-mapfile to verify that a user has access to the resources. For authentication and delegation, GSI

uses X.509 proxy certificates [16] [17].

Using X.509 proxy certificates in a Grid has several advantages [14]. A user can delegate his or her right to other entities on his or her behalf. Also, the mechanism enables single-sign-on for the user to avoid multiple authentication steps to access the Grid resources.

For the single-sign-on, a proxy certificate signed by using the user's private key is generated with a short-term public and private key pair [14]. The certificate contains the short-term public key, and it is stored with the short-term private key [14]. The public key in the certificate is used for the signature validation if the private key is used to sign another certificate for delegation [19]. Likewise, when a certificate is requested from another party for delegation, a public and private key are generated on the party and used for the certificate which is signed by using the private key stored with the user's certificate [14]. In addition, the Certificate Authority (CA) must sign the user's certificate, so that the party can trust its signed certificate [19].

1.2 Cyberaide Shell

Cyberaide Shell [4] has been in the development to lower the difficulty of access to a Grid infrastructure. It provides semantic objects and commands that are easier for users to remember and use. It also furnishes the developers with various language interfaces such as in Java, Python, and JavaScript.

As depicted in Figure 1, Cyberaide Shell [4] can integrate the various cyberinfrastructures by the Plug-ins and provide the services at a higher level by abstracting each of the services. Without dealing with the complex commands and architectures of the toolkits such as the Globus Toolkit [11], users can expect to use the same functionalities easily by avoiding the direct use of the toolkits.

1.3 Water Threat Management Project

The main goal of the threat management research is twofold; characterization of the contaminant source and control of the contamination in WDSs [1]. Currently, the research is focused on the

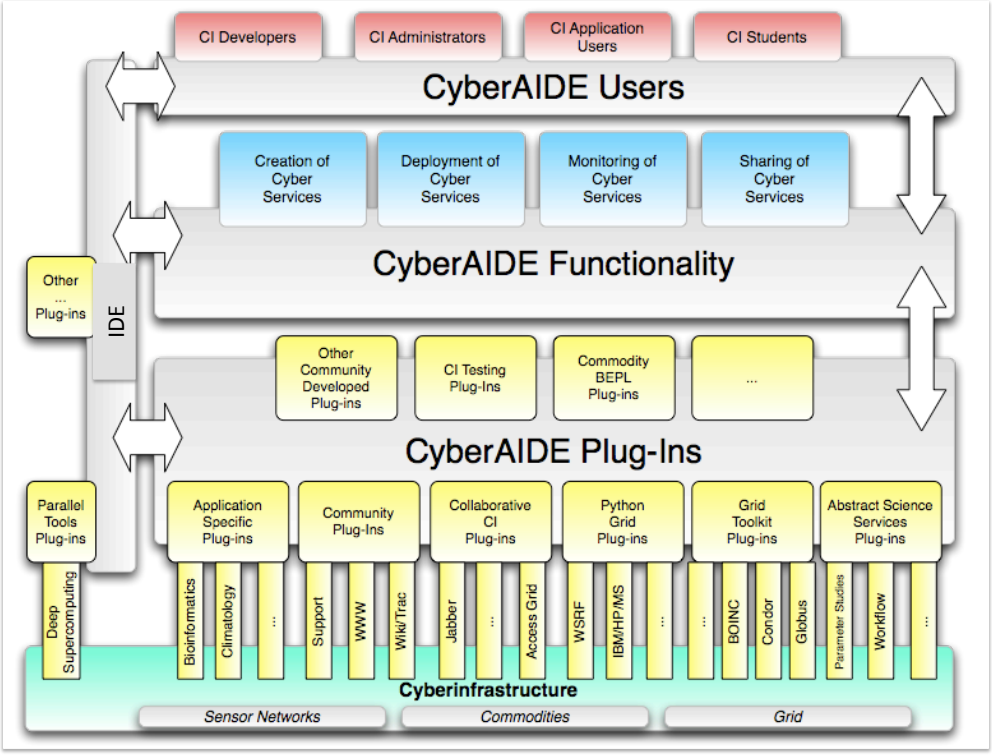

Figure 1: Cyberaide Shell Architecture

source characterization problem [1].

The Water Threat Management system [1] [2] incorporates a Grid middleware, Grid resources, simulation algorithms, sensor networks, a workflow, and a user portal. Figure 2 shows the architecture of the system. Any detection of contamination from the sensors located across the water distribution is sent to the optimization engine. The process in the optimization engine includes an Evolutionary Algorithm (EA) [20] to find an optimized solution for the source characterization by the generation process. The arrows between the optimization engine and the simulation engine indicate that they communicate with each other for the generational processing. In the simulation engine, by the use of EPANET [21], the hydraulic movement of water and change of the water quality is simulated. These processings take place on Grid resources through a middleware with an adaptive workflow for the real-time communication with the sensors or other input data. The workflow can be manipulated through the portal for the simplified management and easier access to the resources.

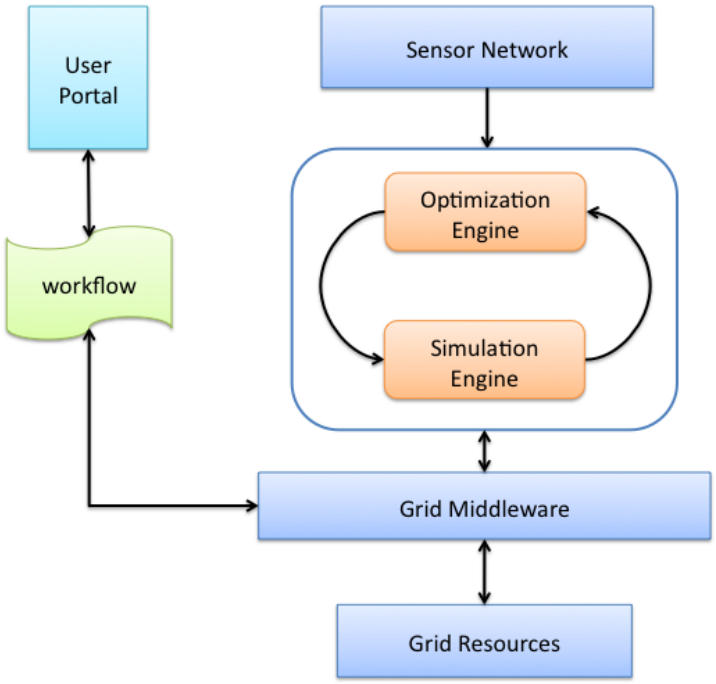

Figure 2: Water Threat Management System Architecture

Additionally, the simulation processing requires large computational power. Nevertheless, the original system [1] failed to satisfy this demand as compared to the later version of the system [2]. The performance of the newer system was improved by the integration of Message Passing Interface (MPI) [22] into the simulation engine so that it parallelizes the EPANET [21] simulation [2]. Figure 3 shows the parallelized EPANET execution in the Water Threat Management system [2]. The source parameters generated in the optimization engine are sent to the master processor (processor 1) in the simulation engine as a text file. The source data is then distributed for the parallel EPANET simulation, and the evaluated results are gathered by the master processor and sent back to the optimization engine. The output file is also a text format. This processing is repeated until it reaches the generation defined in the input parameter of the application.

However, this improved system also has a weakness in that the system is not fault-tolerant, and a failure may cause a fatal problem in society since the threat management requires real-time responses. For that reason, we are developing a fault-tolerant framework and integrating it into the

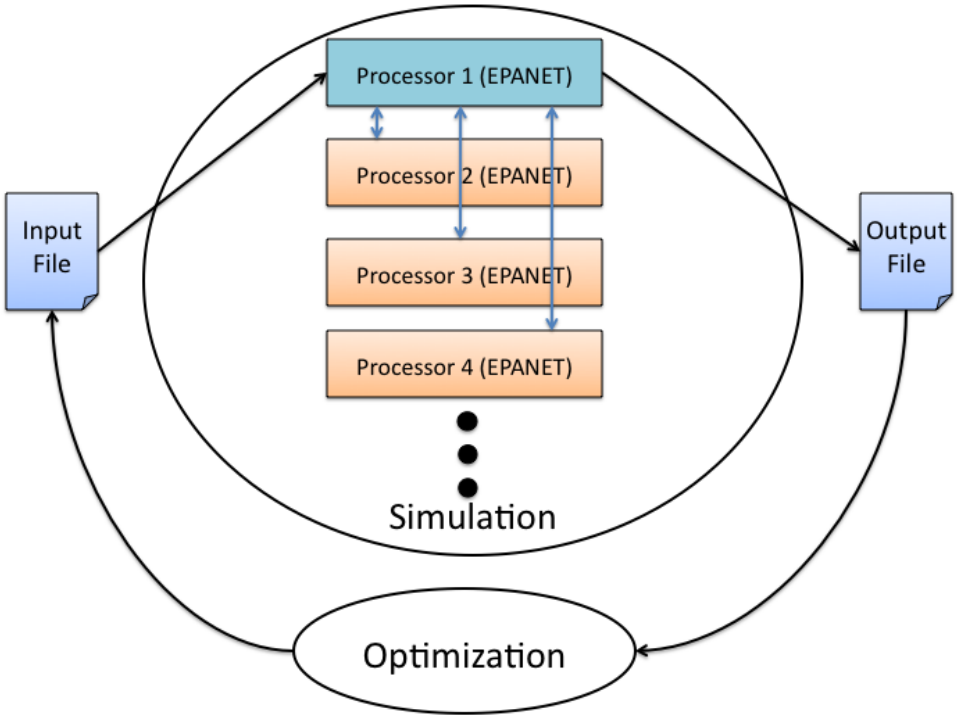

Figure 3: Parallel Execution of EPANET in the Water Threat Management system

existing threat management framework.

1.3.1 Water Threat Management Requirement

The Water Threat Management system [1] [2] requires several aspects to successfully deploy the system as follows:

- *Time-criticality* - As it is a threat management system, this is the most important requirement in the system deployment. The delayed results may cause more costs and challenges to the management of the contamination incidents as the contaminant sources would spread more and wider in the water distribution system as time passes.

- *Massive computational power* - The simulation in the system needs a large amount of calculation. Thus, it requires dedicated compute resources with high performance.

- *Adaptation to the dynamic Grid environment* - Since the Grid environment changes over time, the deployment should be properly adapted to the dynamic environment, so that the simulation can take place on the available resources in real-time.

- *Fault tolerance* - The dynamic feature of a Grid also includes a faulty environment. If the system is not fault-tolerant, all the requirements described above are very hard to achieve when a failure occurs. Therefore, a fault-tolerant mechanism should be integrated into the system.

2 Motivation

Since failures occur frequently in Grid systems [23] [24], the failures can be a significant issue in the deployment of scientific applications necessitating time-criticality. In the case of real-time delivery of service, a failure during the computation may lead to the failure of the deployment of such applications without fault tolerance. Hence, the development of a fault-tolerant framework for the issue is an integral part of the successful deployment of such applications.

In such deployment, the critical factors are the resource availability and queue wait time; it is obvious that resources should be allocated for the jobs for the deployments and queue wait time may cause a long waiting time before the jobs' executions. Thus, these issues should be primarily considered in deploying time-critical applications in the dynamic environments. However, the major obstacles in resolving these problems are the frequent site outages and the uncontrollable queue wait time. In this paper, we consider site outage as the state in which jobs cannot run or may fail due to the scheduled system maintenance, hardware failure, power outage, cooling problems, etc., so that the entire compute resources on a site are not available or very limited for access.

The site outages limit job execution on a specific site for a certain period of time, so that it highly restricts the real-time response when running jobs on the site.

Figure 4 shows the service outages on the TeraGrid [3], including scheduled maintenance and unexpected resource outages due to system failures during 2009 [25]. We include outages of resources providing the GRAM [26] service since job deployments are manipulated through the communication with the GRAM hosting nodes within the Cyberaide Shell [4]. In the graph, the red bars indicate the duration of service outages. In Table 1, the outage rate for each site during the time of 2009 is shown with the number of outages [25]. (Note that we consider the Cobalt site as three different sites: co-compute1, co-compute2, and co-compute3 since each of the machines contains 480, 512, and 256 processors, respectively, and is managed separately.) We categorize outages into two different types: scheduled maintenance and system failure. The scheduled maintenance is a periodical maintenance performed by the site administration, and it is typically announced a few days before the maintenance date. The system failure indicates unexpected outages includ-

ing power outage, cooling system problems, hardware crashes, network service problems, local scheduler problems, file systems failures, storage device failures, login nodes failures, etc., so that users may be affected in running jobs, or running jobs can be lost. On average, there was a service outage approximately 4.7% of the time during 2009, and unexpected outages were approximately 3.2%.

This outage rate can be in fact considered as reliable, however, even a very small percentage of failure should not be acceptable in a threat management.

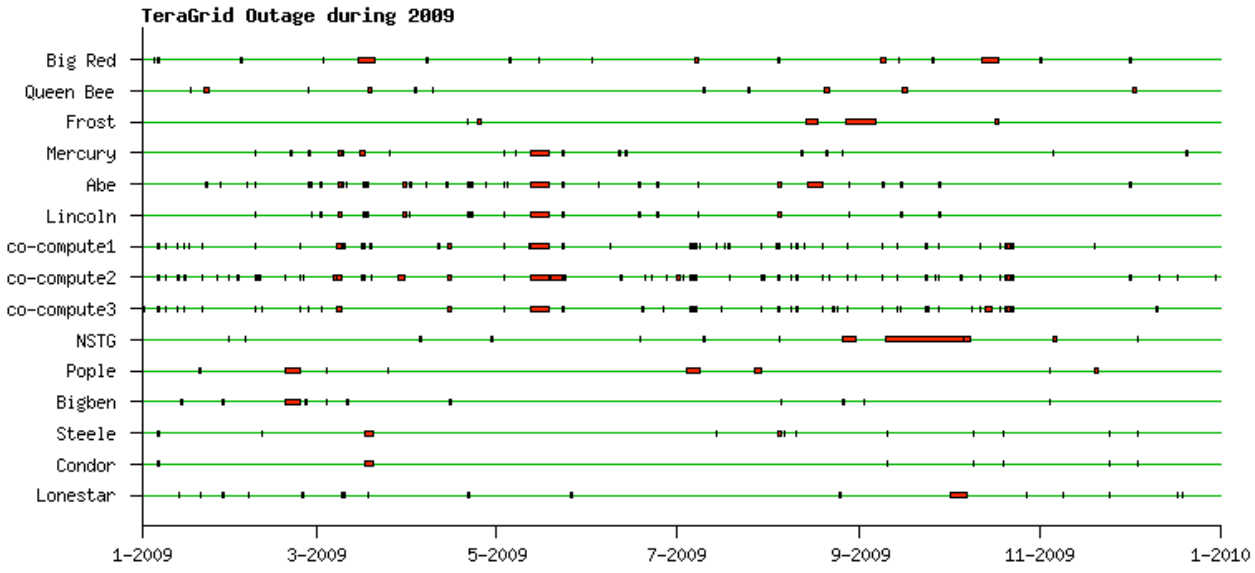

Figure 4: TeraGrid Sites Service Outages in 2009

In addition to the outages, the queue wait time may also lower the quality of service in terms of real-time response. The statistical data of queue wait time shown in [27], of which the experiments conducted in LHC Computing Grid (LCG) [28], indicates that the queue wait times differ greatly. This problem should also be managed properly to increase the quality of service.

The development of our fault-tolerant queue and dynamic workflow are motivated by those issues to launch the Water Threat Management system [1] [2] in a fault-tolerant and dynamic manner in the changing environment to fulfill the requirement of time-criticality of the threat management. The fault-tolerant queue provides dynamic job submission strategies that schedule jobs on multi-

Table 1: Outage Rate (total outage time / year) and the Number of Interruptions on TeraGrid during 2009

Site Name	Scheduled Maintenance	System Failure	Total	#interruptions
Big Red	0.032	0.018	0.050	17
Queen Bee	0.013	0.011	0.024	11
Frost	0.037	0.011	0.048	6
Mercury	0.007	0.031	0.038	21
Abe	0.009	0.056	0.065	38
Lincoln	0.008	0.034	0.043	21
co-compute1	0.012	0.047	0.059	48
co-compute2	0.027	0.062	0.089	65
co-compute3	0.012	0.049	0.061	47
NSTG	0.012	0.092	0.104	13
Pople	0.009	0.031	0.039	8
Bigben	0.007	0.015	0.022	11
Steele	0.015	0.002	0.018	12
Condor	0.011	0.001	0.012	7
Lonestar	0.007	0.020	0.027	17

ple sites to increase the usability of the Grid resources with reliable deployments. Furthermore, although a framework such as QBETS [29] has been developed to predict the queue wait time, it cannot actually affect the queue wait time once a job is scheduled. Our workflow design for the Water Threat Management application [2] actively utilizes the queue wait time to possibly "reduce" it. In addition, the queue is designed for the easy use of the framework and access to the resources as it is built in the Cyberaide Shell [4] which helps users to run their jobs with simple interfaces.

3 Problem Definition & Research Objectives

In this thesis, we investigate the following problems in Grid environments to deliver the services in real-time.

- Failures in Grid environments - Failures in Grid systems are common events [23] [24]. In our framework, we also resolve a site outage problem described in Section 2. It may cause the significant delay or failure of the services.
- Queue wait time - The jobs submitted to run on Grid resources are scheduled by the local batch queuing systems. However, the scheduling may delay the job execution for hours. This might be undesirable for the users and may lower the quality of services for time-critical problems.

To solve those problems in the deployment of the Water Threat Management system [2], we modify the application with newly defined issues within the application deployment. Before the discussion, we first analyze parallelism in the Water Threat Management application [2].

3.1 Parallel Domain Decomposition of the Water Threat Management Application

The simulation in the Water Threat Management [2] deployment can be parallelized in several different ways. As shown in Figure 3, the individuals, which are the source parameters for the EPANET [21] simulation, are parallelized in the simulation engine, and this parallel execution is done at each generation (see Figure 5). This parallel processing can be done within a single job execution using MPI [22], or the individuals can be distributed on multiple job executions on the different resources. In the latter case, each job can process the individuals either with parallelization or without it.

Depending on the number of individuals for the simulation, how to parallelize the individuals may have a different effect on the total run time. Since the number of processors for parallel

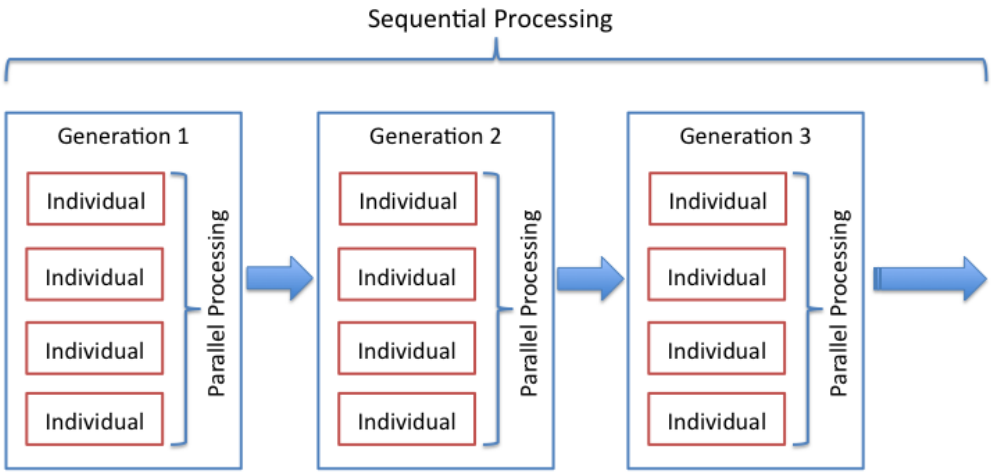

Figure 5: Parallel and Sequential Processing in the Water Threat Management system

processing for a single job is limited by each site policy, the parallel execution within one job has a limitation in which the larger number of individuals (than the number of processors allowed) cannot be fully parallelized. This aspect causes some of the individuals to be processed sequentially. If multiple jobs, each of which runs the individuals in parallel, are deployed, the portion of sequential execution can be reduced. However, this causes another problem that each of the jobs may have a different queue wait time, so that the individuals may not be processed in parallel completely. Therefore, it is not certain which way would result in a smaller run time. In addition, managing the parallelized individuals with multiple job deployments requires more complex system architecture with no significant advantage.

3.2 Modification of the Water Threat Management Application

The following issues are considered in the modification of the Water Threat Management application [2] with its deployment.

- Parallel vs. Sequential processing - The Water Threat Management simulation [2] involves both parallel and sequential processing; parallel execution of EPANET [21] and the generation processing by EA [20] respectively (see Figure 5). In this thesis, however, only sequential processing of fault tolerance is considered in our dynamic workflow. The parallel processing in our system is the same as the system in [2]. Figure 6 shows what is changed in our workflow deployment. In [2], the generations are computed in a single job. However, in our dynamic workflow, the generations are divided and computed as multiple jobs.

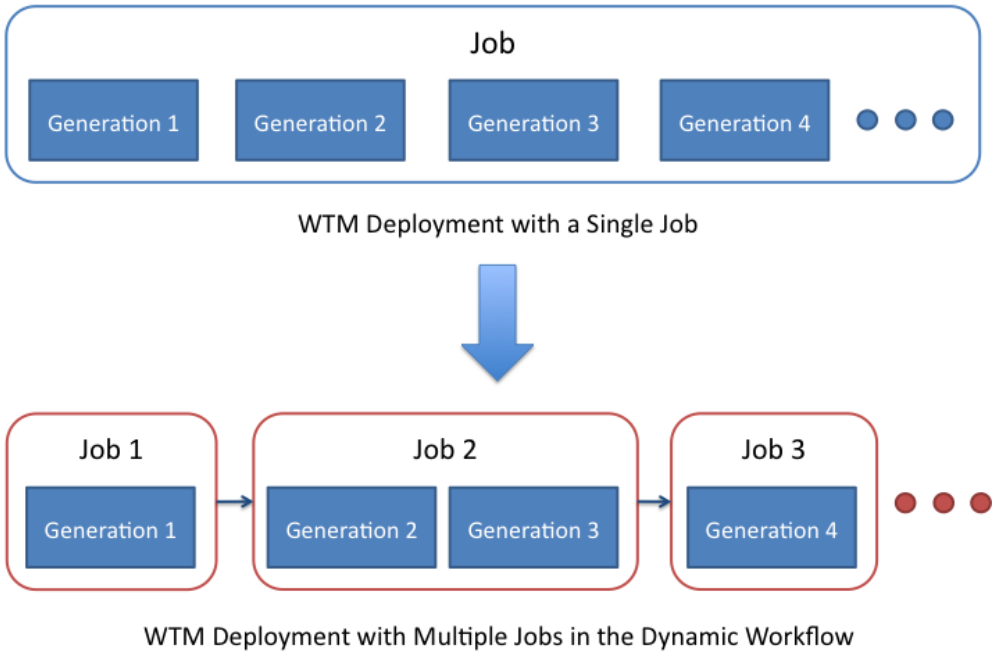

Figure 6: Modification of the Deployment in the Sequential Processing of Generations: The generations are divided into multiple parts as multiple jobs in the dynamic workflow. It shows an example of the distribution.

To divide generations and run them as different jobs as illustrated in Figure 6, we modify the application [2] such that the data of individuals and populations that are being computed in the memory during the simulation can be stored into a text file after the completion of the specified number of generation processing (Figure 7). The file can be provided as an input file to the job starting from the next generation. Since this processing requires only saving

the data into a file at the end or loading the data at the beginning of the job execution, it does not cause big overhead. Also, using a text file format gives compatibility between different system architectures.

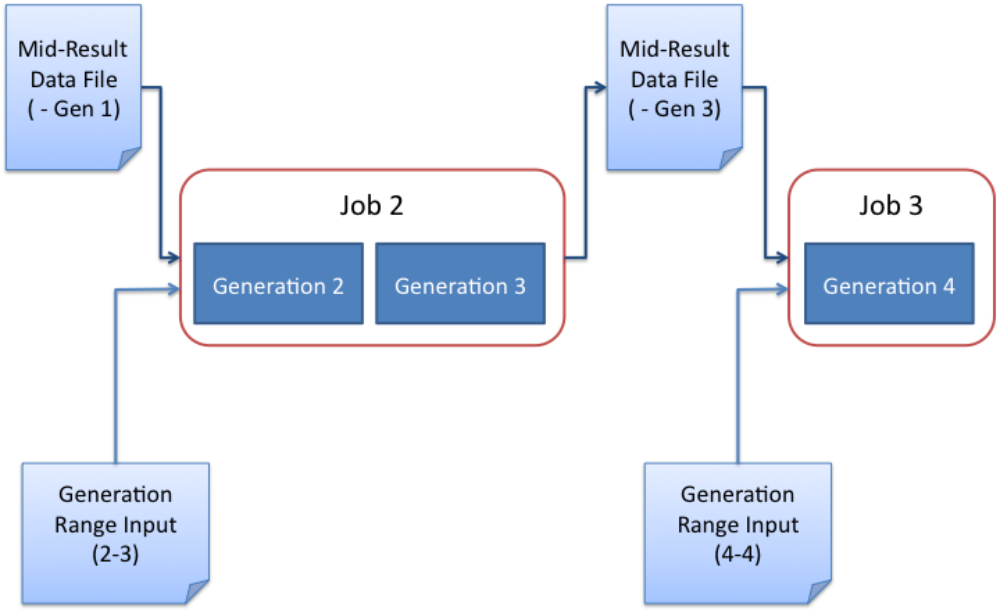

Figure 7: Input and Output Files for Multiple Jobs in the Dynamic Workflow

- Distribution of the generations - To avoid the service interruption from a single site, we distribute the computation of the generation on multiple sites with deployments of multiple jobs. The generation range for a job is supplied as a text file, so that the job can compute the specified range of generations (Figure 7). As we deal with sequential computation of the generations, this introduces job dependency in the deployments.

- Job dependency - Suppose the scenario that there are three jobs, A, B, and C with dependency such that C is dependent on B, and B is dependent on A as shown in Figure 8. Then, C cannot start before B is done, and B cannot start before A is done. In other words, C cannot be submitted to a queue before B is done, and B cannot be submitted before A is done since B may start before A is finished if B is submitted before A is done, and so on. This

problem may not be an issue if the jobs are scheduled at the same local resource manager, however, since our strategy is submitting jobs on multiple sites, it is not considerable in our framework. Therefore, the queue wait time can be a big portion of the deployment of the application.

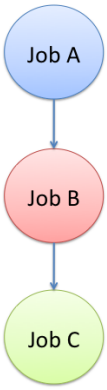

Figure 8: An Example Job Dependency

To solve this issue, our approach is determining the job dependency after job submission, depending on the job start. Figure 9 and 10 show two cases of job start. In the first case (Figure 9), job B starts earlier than job A, so the dependency is violated. Since it cannot be predicted when a submitted job will start, job B should be submitted to Site 2 after job A starts, and this will increase the total queue wait time for the job executions. However, in the second case (Figure 10), the job dependency is not determined at the time of job submission to the queuing systems on the sites. Once any of the jobs starts, the dependency can be determined by assigning tasks for job A to the started job, so job A can start earlier than job B. This method can reduce the total queue wait time and adapt the workflow well to the changing environment.

In addition, to enable such deployment with the dynamic job dependency, we develop a fault-tolerant queuing system and integrate it into the Cyberaide Shell [4] which has an interface of job submission to the TeraGrid [3]. We also develop a workflow engine for the dynamic scheduling for an automated control of the workflow.

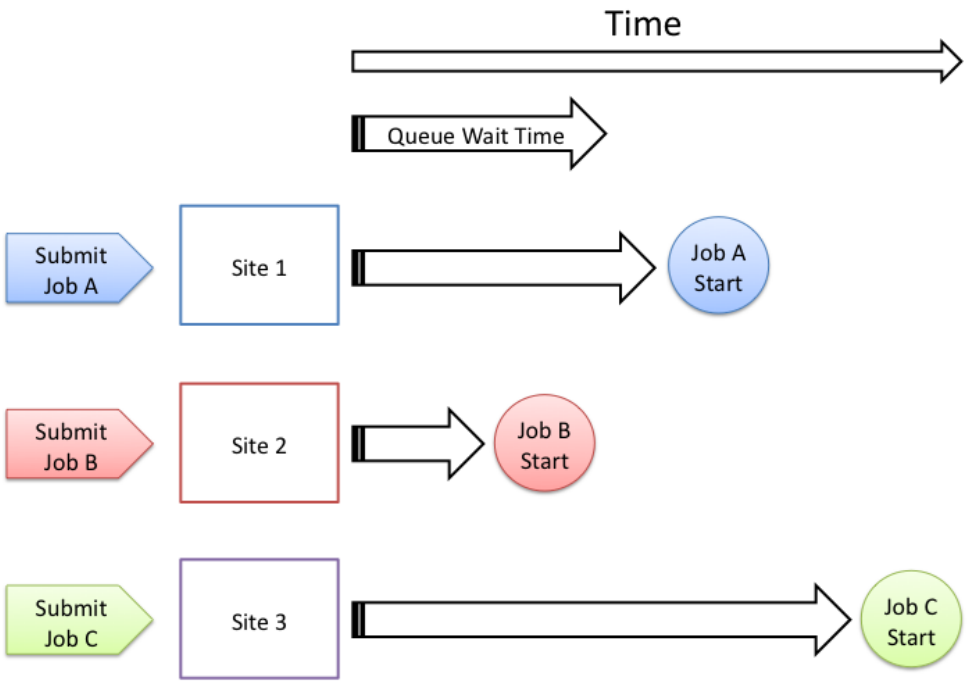

Figure 9: Case 1: Job Start without Dynamic Job Dependency

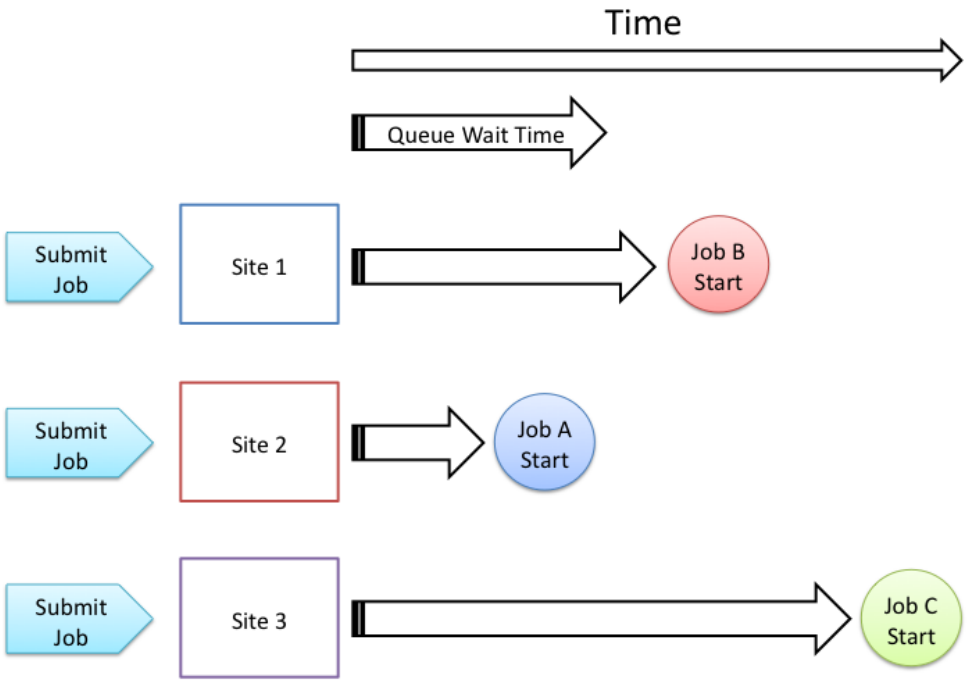

Figure 10: Case 2: Job Start with Dynamic Job Dependency

4 Fault-Tolerant Queue

One of the major parts in building the fault-tolerant workflow is the development of a fault-tolerant queue.

4.1 Background

Before a discussion on the details of the queue, batch queuing systems and the resource management are briefly described as a background of this section.

4.1.1 Batch System

To run a job on Grid resources, the job first needs to be submitted to a local resource manager (LRM) such as PBS [9]. It schedules user jobs on the resources with the information on the job description written in a script file. The user can specify the name of the queue, wall clock time, memory size, number of processors, and so on in the job description. The resource manager allocates the resources for the job when the job can be executed with the requirements provided by the user.

4.1.2 Job Submission & GRAM

There can be several ways to submit a job to the queuing system to run the job on the compute nodes. An example is that a user logs into the login node on the site via *gsissh* [30] or a portal such as [31] and runs the command provided by the site. An alternative method is that a user runs the job submission command provided by a Grid toolkit such as the Globus Toolkit [11]. While the former may provide a distinct interface to submit a job due to the different LRM systems, the latter provides a uniform interface to job submission since the Globus Toolkit [11] integrates the different LRMs into its architecture and provides the interface at a higher level. Figure 11 depicts job submission through GRAM [26] on a single site. The user jobs are in the local disk on the site, and the gatekeeper node hosts the GRAM service. When a user submits the job through GRAM,

the GRAM service sends the job to the LRM and communicates with it to manage the job. The LRM actually allocates the compute node for the job and stores the output in the local disk.

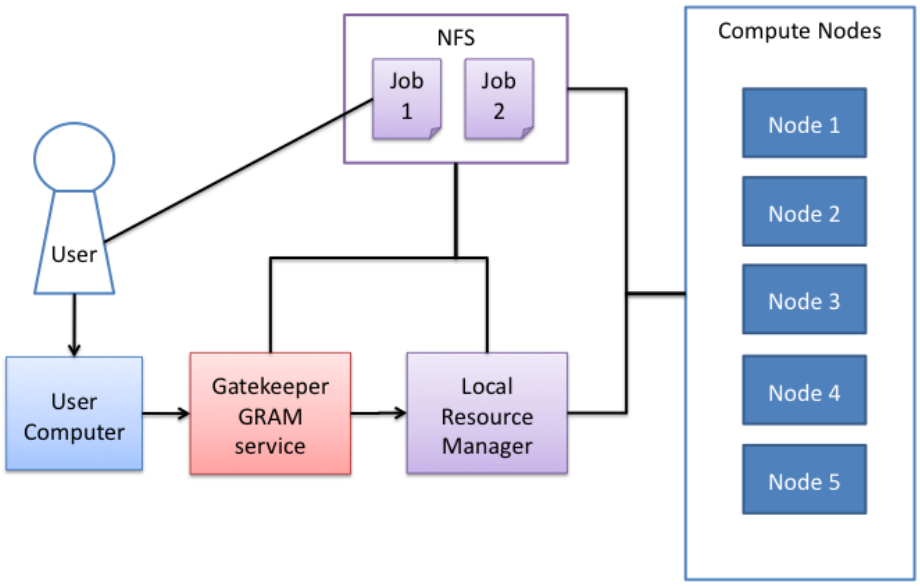

Figure 11: Job Submission with GRAM

For the job submission through the fault-tolerant queue in Cyberaide Shell [4], GRAM [26] of the Globus Toolkit [11] is used to manage jobs on the different sites. The main functionalities of GRAM are job submission / execution with batch or interactive mode, monitoring job states, and file staging [11]. These functionalities are provided as Web Services [12] [11]. Rather than just exposing the same functionalities to the users, the queue internally uses some of the services such as monitoring job states, job submission, and cancelation for the fault-tolerant job scheduling within the system.

4.2 Need for Fault-Tolerant Queue

As described in Section 2, there were outages about 4.7% of the time during 2009 on the TeraGrid [3] sites [25], and this may restrict the deployment of time-critical systems with the limited

resource availability. Moreover, 3.2% of outages was unpredictable [25], which would cause an unavoidable failure of system deployments.

In such environments, some of the current fault-tolerant mechanisms may be insufficient to successfully deliver the real-time services; it highly restricts the migration of a job and makes checkpointing impractical in terms of time-criticality since the jobs cannot be rescheduled on the same site while the site is down. In addition, the threat management problem is not only a matter of fault tolerance but also a matter of reducing the time from detecting the contaminations at the sensors to delivering the final computational results. Therefore, the fault tolerance with this threat management should avoid focusing on only a local-scheduling strategy. This requirement can be fulfilled by the fault-tolerant queue that distributes jobs on the multiple sites and schedules them in a fault-tolerant manner.

4.3 Fault Tolerance Strategy

For an application to be fault-tolerant, it is obvious that all the computations need to be done correctly regardless of a failure. Since it is very difficult to predict a crash perfectly, the basic strategy of fault tolerance is trying to rerun a program until it results in the success of the execution. The most common fault tolerance strategies used in a Grid are replication and checkpointing [32]. While replication runs a replica in addition to the original job, checkpointing operates like the "continuous shooting mode" of a digital camera on the state of a running job [33]. This reduces the portion of the duplicated computation in case of a failure but decreases the performance due to the checkpointing process [32] [33].

In the consideration of applying fault tolerance into our framework, as discussed in Section 4.2, checkpointing cannot be used effectively. Moreover, the different sites may be heterogeneous; the resources on the different sites may have different platforms and software, so the data of checkpoint on one site may be incompatible on the other site.

The fault-tolerant strategy that is suitable to our model should not be system-dependent and should be integrated into the framework with high performance. Thus, replication is the proper

method that satisfies those conditions for our fault-tolerant queuing strategy.

4.4 Design

In this section, the design of the fault-tolerant queue and its implementation are described.

4.4.1 Architecture

The queuing system is composed of several components: a task pool that keeps user jobs submitted to the pool, a resource pool that has information about each of the TeraGrid [3] resources, a scheduler that submits a job from the task pool after mapping the job to a resource, and a resource checker that detects resource unavailability. Figure 12 shows the overall architecture of the system. Users can submit their jobs to the task pool via the Command Line Interface (CLI). The scheduler maps a job in the task pool to a resource in the resource pool by the policies defined in the system to run the job on the TeraGrid resource. The job submission to the TeraGrid resources is done via the GRAM [26] service on each site as the Cyberaide Shell [4] services intercommunicate with the components of the Globus Toolkit [11].

In addition, the resource elements in the resource pool change over time; the resource checker periodically retrieves information about resource availability from the TeraGrid Information Services [34] and adds/removes the elements dynamically. Thus, user jobs are not likely submitted to the unavailable resources when they are scheduled. This site outage information checking mechanism also has another purpose to detect a job failure caused by the site outage. It is described in detail in Section 4.4.5.

4.4.2 Policies in the Queuing System

A job and a resource selection from the pools are decided by each policy on the task pool and the resource pool. By default, the system has the First In First Out (FIFO) policy for the task pool and the Round-Robin policy for the resource pool. In the FIFO policy, a job submitted first to the task pool is selected first. In the Round-Robin policy, a resource is selected based on taking a

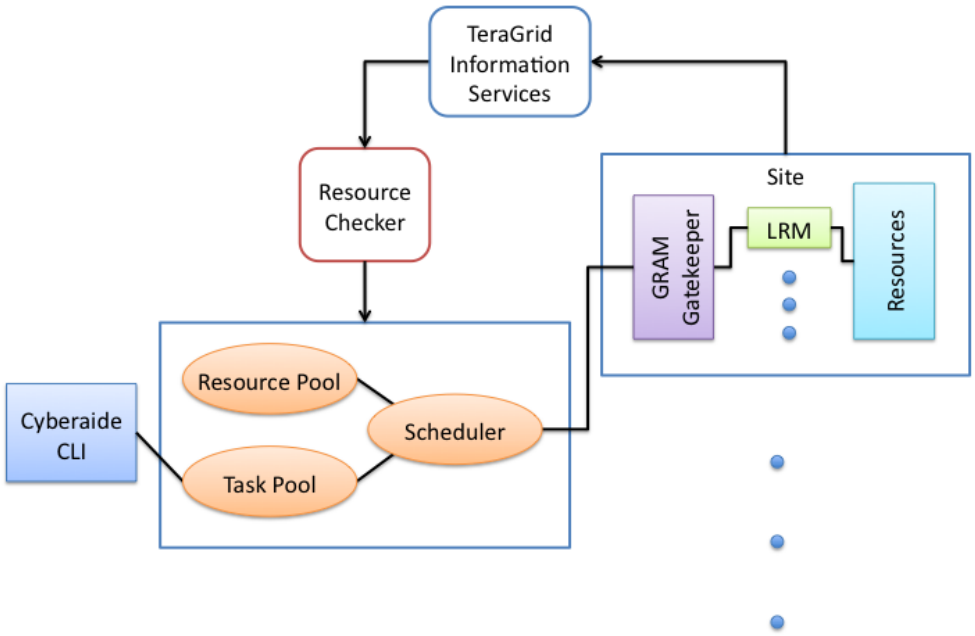

Figure 12: Architecture of the Queue of Cyberaide Shell

turn. Additionally, more policies can be designed and added to the system, and the policies can be changed via CLI.

Fault-tolerant job scheduling can be done by setting the replicate policy to the task pool. Under the replicate policy, the job selected by the scheduler is replicated, then the replicated job is submitted to the TeraGrid [3] resource while the original job remains in the task pool. If the replicated job fails, the job can be selected and resubmitted. If the submitted job is finished successfully, the scheduler removes the original job from the pool.

4.4.3 Command Line Interface

The Command Line Interface (CLI) is designed for the simple and easy use of the queuing system. The configuration of the Grid system such as hostname and endpoint reference (EPR) can be supplied as text files, so it can simplify the CLI for the users. Table 2 lists some of the subcommands of the queue command, and the detail usage is described in Appendix A.

Table 2: Cyberaide Shell Queue Commands

Command	Description
-submit	Submit a job defined by a user.
-stat	Display the job status.
-suspend	Suspend a job in the queue.
-resume	Resume a job in the queue.
-cancel	Cancel a submitted or running job.
-policy	Set a policy.

The following is an example of job submission with a fault-tolerant policy.

```
cybershell> queue
queue> policy -task -replicate
queue> submit -cmd /mydir/job
```

After setting the replicate policy to the task pool, every job in the task pool is replicated when it is scheduled to run. The jobs are removed from the pool after the successful executions as described in Section 4.4.2. Also, users do not need to supply the full address of EPR in the command for a remote job execution since the resource can be selected and the corresponding EPR informaiton can be supplied within the system.

The following is another example of running an MPI [22] job in the CLI.

```
queue> submit -cmd /mydir/mpijob -mpi 8
```

Users can simply set the *-mpi* option with the number of processors to execute the program. The system finds the necessary information to decide the number of nodes or the right version of the *mpirun* program to run the job based on the different resource architecture if it is configured within the system. As the Water Threat Management [2] is an MPI program, it can be deployed with the simple command as shown in the example.

4.4.4 Lifecycle of a Job in the Fault-Tolerant Queuing System

A lifetime of a job can consist of several states. The queue defines the states of a job as *submitted*, *canceled*, *running*, *done*, and *failed* as described in Table 3.

Table 3: Job States in Fault-Tolerant Queue

State	**Description**
Submitted	A job is in the pool.
Canceled	A job is canceled
Running	A job is running on the resource
Done	A job execution is finished
Failed	A job is failed

In the queuing system, there can be several different cases of the lifecycle of a job. The lifecycle can be affected by the service availability, job status, and the policy. The transition diagrams in Figure 13 and 14 show the lifecycle of a job under the different policies set to the task pool. The user intervention such as canceling a job by a user is not considered.

Figure 13 shows the lifecycle of a job in the system. The job status can change from the *running* status to either the *failed* or *done* status. The job status can also change from *submitted* to *failed* directly when the job submission fails due to the failure of the GRAM [26] service on the gatekeeper node.

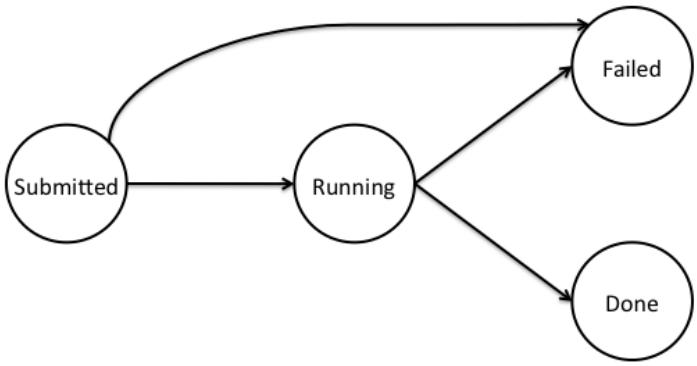

Figure 13: Lifecycle of a Job under the Default Policy

The lifecycle of a job under the *replicate* policy is shown in Figure 14. The lifecycle of a job is basically identical to the one shown in Figure 13. However, the *failed* status can change to *submitted* if the execution fails. Also, the change of the job status from *failed* to *submitted* can be repeated in the lifecycle in case of job failure while the task policy is set to *replicate*.

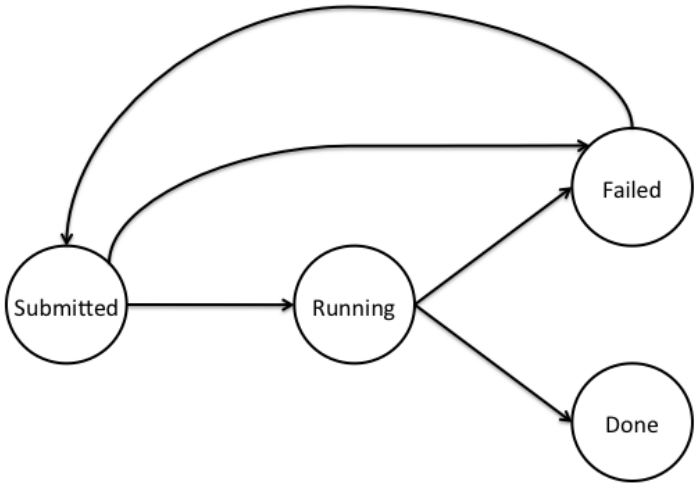

Figure 14: Lifecycle of a Job under the Replicate Policy

4.4.5 Fault Detection & Information Services

One of the important functions of this queuing system is fault detection. The primary method to detect a job failure is catching the message from GRAM [26] of the Globus Toolkit [11]. However, there is a case where GRAM may not detect a job failure correctly. Since GRAM communicates with the local resource managers (LRMs) to check the job status if the job is submitted through the LRM, the failure of the LRM may supply inaccurate information.

This failure of LRMs is certainly an obstacle for the fault-tolerance service in our framework. Instead of relying on only the GRAM service, we incorporate an additional method for the fault de-

tection. The fault-tolerant queue integrates the TeraGrid Information Services [34] into the system to deal with the case of a failure of the LRM. TeraGrid [3] publishes various information related to software, resources, a number of jobs on the queues and resources, and the site outages in a number of ways. The users can get the information through the user portal [31], Monitoring and Discovery System (MDS) [35], or the web applications [34]. The resource checker in the queue retrieves the site outage information which is published as XML documents from [34], so it can be directly used to manage the job failure in the queuing system. If the resource is down due to the site outage, the queue detects the outage and sets the job status to *failed*. This method can assist to detect the job failure in the case that the LRM malfunctions.

5 Formal Analysis for Fault-Tolerant Workflow System

In this section, we describe a workflow for a Grid and the model with a theoretical approach to obtain the run time.

5.1 Workflow

Essentially, a workflow is an arrangement of a number of tasks that need to be done in sequential order, non-sequential order, or combination of both. However, in Grid environments, several other factors should be involved in the workflow in addition to the tasks themselves. As described in [36], a workflow involves not only an arrangement of jobs but also resources, services and quality expectations from the users. These elements can change over time, and the workflow is affected by them to meet the user expectations or be adapted to the dynamic environment.

As defined in [36], a Grid workflow can be formally expressed as

$$W_i = (G_r, G_s, Q_u, W_m),$$

where

W_i represents workflow instantiation;

G_r represents Grid resources;

G_s represents Grid services;

Q_u represents Quality expectations from the user;

W_m represents Workflow model.

Also, the dynamic change of workflow is defined in [36] as

$$W_i^T = (G_r^T, G_s^T, Q_u^T, W_m^T)$$

$$\downarrow a^T$$

$$W_i^{T+1} = (G_r^{T+1}, G_s^{T+1}, Q_u^{T+1}, W_m^{T+1})$$

where a^T is an adaptation function at time T. As shown in this formula, a Grid workflow can change over time to adapt to the dynamic environments.

5.2 Run Time Analysis for Dynamic Job Dependency

Although distributing jobs on multiple sites can avoid the loss of all the jobs or the "delay" of scheduling jobs on a single site by the site crash, it may cause an additional queue wait time when jobs are submitted to the local schedulers on multiple sites. To reduce this side effect, we integrate the dynamic job dependency strategy into our framework.

Suppose that there are n jobs from j_1 to j_n, and job j_k is dependent on j_{k-1} where $1 < k \leq n$ such that job j_k cannot start before j_{k-1} finishes, and queue wait time q_k corresponds to job j_k. Then, job j_k cannot be submitted to the local scheduler before job j_{k-1} finishes since job j_k may start before job j_{k-1} finishes. The total run time including queue wait time can be formally expressed as

$$T_t(n) = \sum_{i=1}^{n}(q_i + T_s(j_i)) \tag{1}$$

where T_t denotes a time from the submission of job j_1 to the finish of job j_n, and where T_s denotes a run time of a single job.

$T_t(n)$ can be reduced if job j_k can be submitted before job j_{k-1} finishes and j_k waits starting its computation until j_{k-1} is done. Assume that all jobs are submitted at the same time. Then, the run time of a job is

$$T_t(n) = \begin{cases} q_1 + T_s(j_1) & n = 1 \\ \max(q_n, T_t(n-1)) + T_s(j_n) & n > 1 \end{cases} \tag{2}$$

However, (2) is still largely dependent on the queue wait times of the jobs that should be done earlier in the workflow since the performance of n jobs is dependent on the performance of $n-1$ jobs if q_n does not exceed $T_t(n-1)$.

Our approach to this problem is setting job dependency at run time which is described in Section 5.3, so that $T_t(n)$ can be reduced by removing the dependency between a job and its corresponding queue wait time; a job with a longer queue wait time becomes dependent on the job with a shorter queue wait time. This improvement can be formally expressed as

$$T_t(n) = \begin{cases} Q_{min}(1) + T_s(j_1) & n = 1 \\ \max(Q_{min}(n), T_t(n-1)) + T_s(j_n) & n > 1 \end{cases} \quad (3)$$

where $Q_{min}(k)$ denotes the kth minimum value among $q_1, q_2, \cdots, q_n$ $(1 \leq k \leq n)$.

Fundamentally, the computation of the generations of the Water Threat Management [2] system is dependent on each other so that each generation must be computed one after the other. Thus, the application can be altered such that any specified range of the generations is computed as a single program while the application code remains the same. This makes the whole single process divided into multiple processes as multiple jobs with dependencies. Since the code is the same for the computation of the different generations, the job dependency can be determined at run time when each of the jobs starts; A job that starts earliest can be assigned to compute from generation 1, and the next starting job can be assigned to compute the next generations to the ones of the first job, and so on. This can be well fit into the dynamic job dependency model shown in (3).

5.3 Workflow Model for Water Threat Management

The Water Threat Management system [2] computes a number of generations to determine the correct contaminant source locations with the parallel EPANET [21] simulations. Although dividing the generations and parallel executions can be considered all together, we focus on the distributing generations as described in Section 3.

In the model, the number of generations can be divided into several parts as multiple jobs. If the system runs n generations, it can be divided into k parts where n and k are positive integers with $n \geq k$. Each of the jobs in our system can start from generation m $(1 \leq m \leq n)$ which is specified by a user, so that users can flexibly decide the range of generations for their experiment

plans. In addition, this system stores the "mid-results" of the generation into a file to supply them to the next job. Since it cannot start from generation m if $m > 1$ as the first job, the results of generation $m - 1$ must be provided to start from generation m if $m > 1$.

Figure 15 shows an example case of the dynamic job scheduling depending on the different job start times. When the job starts on the resource, the job generates a text file to notify its start. The workflow manager in the system then can detect which job has started on which resource. After that, it generates an input file specifying the generation range and transmits the file to the job. If the started job is not the first job, the "mid-result" file is sent to the job to continue the simulation. The "mid-result" file generated from the job is transmitted to the workflow manager (if the job is not the last job) as an input of the next job. The execution time may vary depending on the resource performance, and there can be a wait time on the job if the preceding job has not finished its simulation.

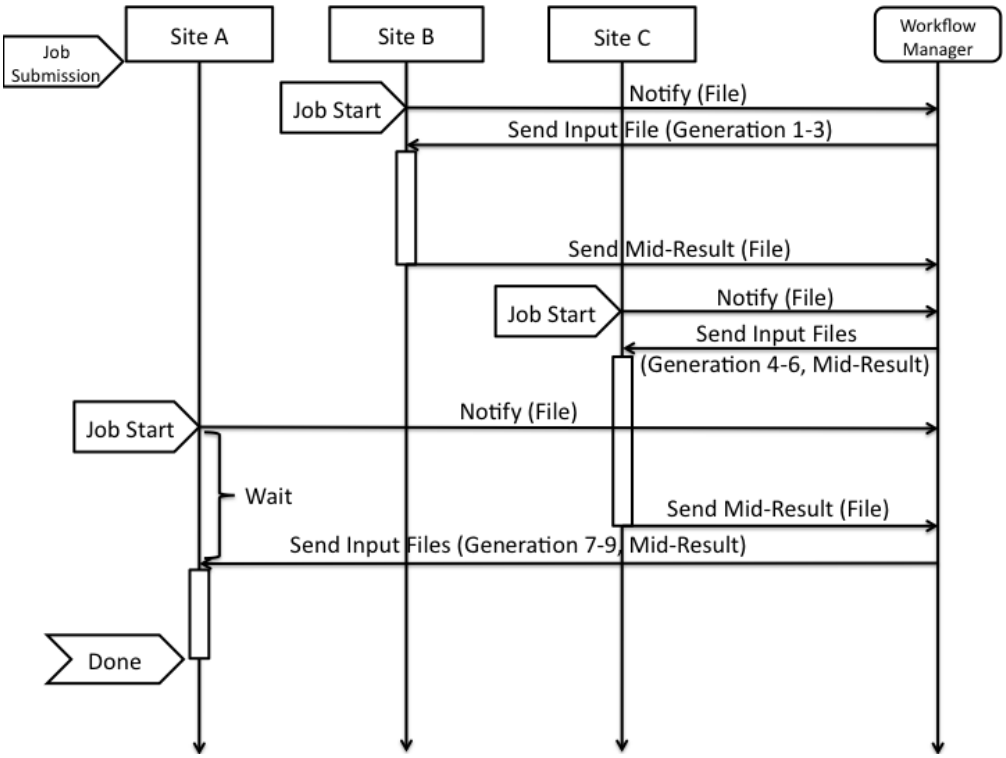

Figure 15: An Example Scenario with Dynamic Job Scheduling for 9 Generations

If a failure occurs in any of the jobs during the execution, the failure is notified in the workflow

manager, and the workflow manager retransmits the input files sent to the failed job to the next starting job. Also, since the processes are sequential and managed in a synchronized fashion, there is no intervention of the input data between the different jobs. Algorithm 1 describes the workflow management with multiple thread executions. Each thread corresponding to each job manages the input/output file transmission between the job and the Workflow Manager with synchronization.

Algorithm 1 Water Threat Management Workflow Control: In the Workflow Manager, a thread is created per job, and it controls the data (file) communication in a coordinated manner.

```
1 wait until job starts
2 synchronized{
3     send generation range input file to the job
4     if (the job is not the first job), then send mid-result file to the job
5     wait until job finishes
6     if (the job is not the last job), then receive mid-result file from the job
7 }
```

With this model, the whole computation can be divided into a number of parts as multiple jobs on multiple sites rather than a single job on a single site, and this distribution can have a better chance to reduce the run time of n generations in case of a failure as discussed in the next section.

5.4 Run Time Analysis for Water Threat Management on Multiple Sites

When failures occur, the run time increases since the failed jobs should restart, so it is worth it to understand how the failures would affect the run time. Our hypothesis is that the job's additional run time caused by failures decreases as a job is divided into more number of parts as multiple jobs, each of which has a run time equal to the original job's run time divided by the number of parts. In this section, we estimate how the run time changes based on the number of sites used to distribute the generations with the consideration of failure. In addition, the queue wait time is not considered in this estimation.

We calculate the run time based on the following assumptions:

- The probability that any single site crashes during a certain period of time t is p.

- The event of failure during an additional period of time t' is independent to the event of failure during t where $t' = t$.

- All machines have the same compute performance.

Then, the probability that a site crashes during $2t$ can be obtained as follows:

Let $t_1 = t_2 = t$, so that $2t = t_1 + t_2$. Then, the probability that a site does not crash during both t_1 and t_2 is $(1-p)^2$. Also, the probability that a site crashes during at least one of t_1 and t_2 is the negation of the probability that a site does not crash during both t_1 and t_2 which is $1-(1-p)^2$, and this is the same as the probability that a site crashes during $2t$.

Likewise, the probabilities that a site crashes during $3t$, $4t$, $\cdots$ can also be obtained.

Consequently, an approximated function for the probability relating to the duration can be derived, and it can be expressed as

$$f(x) = 1 - (1-p)^x, \quad x > 0 \tag{4}$$

where x is the amount of time, and $f(x)$ is the probability that a site crashes during x.

Now, we can calculate the run time by applying the function (4).

Let $T(n)$ be the run time of n generations, and assume that n generations are equally distributed on k site(s) with k job(s) ($k = 1, 2, 3, \cdots$) such that a single job runs n/k generations. The run time of each of the jobs is then $T(n)/k$, since $T(n)$ has a linear relationship to n as shown in Figure 16 in Section 6. In addition, the run time of each of the jobs (including the "failure" run time) can be obtained as follows:

Let X be the number of times to run the job with $\frac{T(n)}{k}$ until the job runs without a crash ($X = 1, 2, 3, \cdots$), and assume that there are only two ways of the event, no-failure of a job and failure of a job. Then, the probability that the job fails (by site crash) during its run time $\frac{T(n)}{k}$ is $f(\frac{T(n)}{k})$, and the expected value of X is

$$E(X) = \frac{1}{1 - f(\frac{T(n)}{k})} \tag{5}$$

since it has a geometric distribution.

Besides, the run time for X can be obtained. Suppose that the time when a job fails during its execution has a uniform distribution *in the case that the job fails*. Then, the run time of a job when it fails is $\frac{1}{2} \cdot \frac{T(n)}{k}$. The following is the run time for each number of trials.

X	**Run Time**
1	$\frac{T(n)}{k}$
2	$(\frac{1}{2}+1) \cdot \frac{T(n)}{k}$
3	$(\frac{2}{2}+1) \cdot \frac{T(n)}{k}$
4	$(\frac{3}{2}+1) \cdot \frac{T(n)}{k}$
	$\vdots$
m	$(\frac{m-1}{2}+1) \cdot \frac{T(n)}{k}$

(6)

From formula (6), we can derive a job's run time function as

$$T_j(m,k) = (\frac{m-1}{2}+1) \cdot \frac{T(n)}{k} \tag{7}$$

where m is the number of trials until the "success" of the job, n is the number of generations, and k is the number of sites $(k = 1, 2, 3, \cdots)$.

So, approximately, the run time of the job with the expected number of trials can be obtained by substituting expected value (5) to function (7) as

$$\begin{aligned} T_j(E(X),k) &= [\tfrac{1}{2}(\tfrac{1}{1-f(\frac{T(n)}{k})}-1)+1] \cdot \tfrac{T(n)}{k} \\ &= [\tfrac{1}{2}(\tfrac{1}{1-[1-(1-p)^{\frac{T(n)}{k}}]}-1)+1] \cdot \tfrac{T(n)}{k} \\ &= [\tfrac{1}{2}(\tfrac{1}{(1-p)^{\frac{T(n)}{k}}}-1)+1] \cdot \tfrac{T(n)}{k} \\ &= \tfrac{1}{2}(\tfrac{1}{(1-p)^{\frac{T(n)}{k}}}+1) \cdot \tfrac{T(n)}{k} \\ &= (\tfrac{1}{(1-p)^{\frac{T(n)}{k}}}+1) \cdot \tfrac{T(n)}{2k} \end{aligned} \tag{8}$$

Therefore, the run time of n generations with $E(X)$ with k jobs on k sites is

$$\begin{aligned} k \cdot T_j(E(X), k) &= k \cdot \left(\frac{1}{(1-p)^{\frac{T(n)}{k}}} + 1 \right) \cdot \frac{T(n)}{2k} \\ &= \left(\frac{1}{(1-p)^{\frac{T(n)}{k}}} + 1 \right) \cdot \frac{T(n)}{2} \end{aligned} \tag{9}$$

This tells us that the run time decreases as k increases, which proves our hypothesis. If k goes to K (K is the maximum number of sites) and K is large enough, the run time is close to $T(n)$, and this can be formally expressed as

$$\lim_{k \to K} \left(\frac{1}{(1-p)^{\frac{T(n)}{k}}} + 1 \right) \cdot \frac{T(n)}{2} \simeq T(n) \tag{10}$$

To verify the efficiency of our workflow, we compare the run time of the dynamic workflow deployment to the run time of the original application [2] deployment.

Since the original application is deployed as a single job on a single site, the run time of the original application is equal to the run time of a job without the distribution. (We assume that the user reruns the job if it fails.) Therefore, the run time of the original one can be obtained from equation (9) where the number of sites $k = 1$ as

$$T_{original}(E(X)) = \left(\frac{1}{(1-p)^{T(n)}} + 1 \right) \cdot \frac{T(n)}{2} \tag{11}$$

where $T_{original}$ denotes the run time of the original application based on $E(X)$ trials and n is the number of generations.

Note that the only difference between formula (9) and (11) is the existence of k. While the dynamic workflow reduces the run time as k increases (see equation (10)), the original deployment has a constant run time if n is a constant.

As discussed through this section, the dynamic workflow deployment within the fault-tolerant queue can reduce the run time of the Water Threat Management application [2] by decreasing the chance of failure by dividing and distributing the generations.

6 Evaluation

In this section, we describe our various experiments with the Water Threat Management application [2].

6.1 Performance of Water Threat Management Application

Prior to developing our fault-tolerant framework, we evaluated the performance of the Water Threat Management application [2].

6.1.1 Objective

The goal of this experiment was to understand the behavior of the application in the distinct system environment.

6.1.2 Test Setup

Due to our limitted access to the TeraGrid [3] resources, we have selected two different machines, Abe and Big Red, for our experiments. The resource description is shown in Table 4, 5, and 6.

Table 4: TeraGrid Resource Description (Hardware - 1)

Site Name	**CPU**	**Interconnect**
Abe	2 quad core Intel 64 (2.33 GHz)	InfiniBand
Big Red	2 dual core PowerPC 970MP (2.5 GHz)	PCI-X Myrinet-2000

Table 5: TeraGrid Resource Description (Hardware - 2)

Site Name	**Total Number of Cores**	**Number of Nodes**	**Total Memory**
Abe	9600	1200	9600.0 G
Big Red	3072	768	6144.0 G

Table 6: TeraGrid Resource Description (Software)

Name	**Operating System**	**Compiler (softkey) used**
Abe	Red Hat Enterprise Linux 4 (2.6.9)	openmpi-1.3.2-gcc
Big Red	SuSE Linux Enterprise Server 9	openmpi-1.3.1-ibm-32

We have tested the performance of the Water Threat Management application [2] on the two different machines based on the number of generation from 1 to 50. The number of cores used for this experiment is 16 for each machine. On Abe, 2 nodes are allocated for 16 cores, and 4 nodes are allocated on Big Red for 16 cores due to the system architectures. The queue wait time was not included.

6.1.3 Result

Figure 16 shows the run time of each of the generations on the Abe and Big Red machines. The run time grows linearly as the number of generation increases.

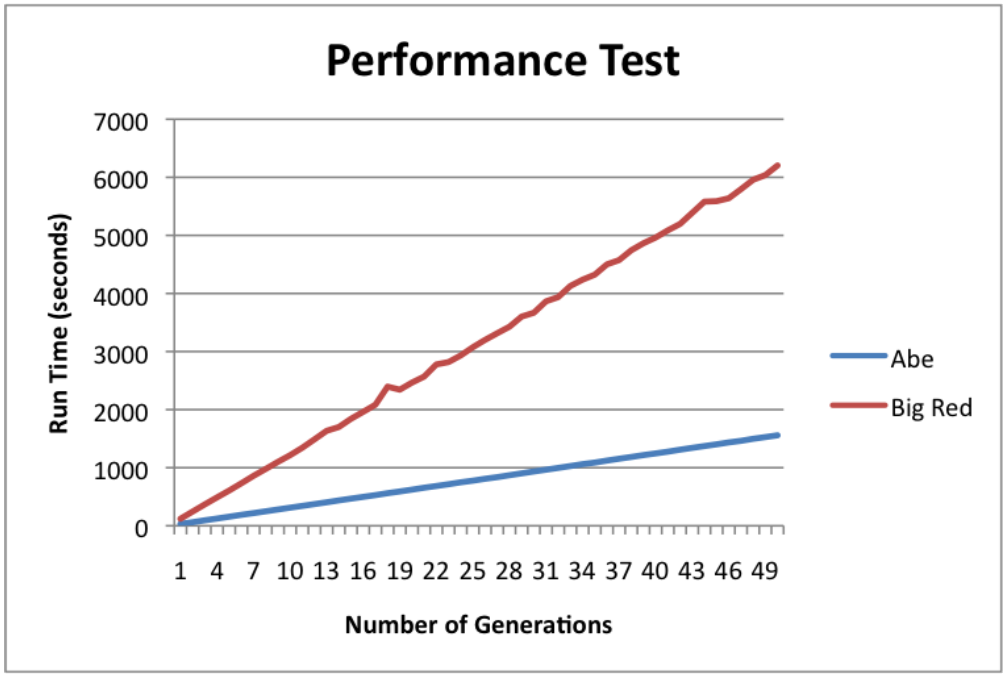

Figure 16: Performance Test with different number of Generation

The run time on the Abe machine is approximately 4 times faster than the run time on the Big Red machine (see Table 10).

6.1.4 Conclusion

From the result, it is expected that the run time of the application can be estimated quite accurately without further experiments. The result can also be applied to model the workflow based on the

different compute performances.

6.2 Queue Wait Time

We have evaluated the queue wait time on the Abe and Big Red machines based on 25 trials.

6.2.1 Objective

The objective of this experiment is to investigate when the resources become usable at the time of job submission for the efficient workflow model.

6.2.2 Test Setup

For each of the machines, one job was submitted per day between 12:00pm and 4:00pm EST, and the two jobs were submitted at the same time for each trial to compare the time difference of the job start. For all of the jobs, the wall clock time was set to 3 hours, and the number of nodes was set to 2 for the Abe machine and 4 for the Big Red machine, respectively. Each job required 16 cores.

Table 7: Queue Wait Time (unit: minutes)

	Abe	Big Red
Average	82	42
Sample Variance	38513	5354
Standard Deviation	196	73

6.2.3 Result

The results in Table 7 are the average, sample variance, and standard deviation of the queue wait time for each of the machines. The histogram of the queue wait time is shown in Figure 17 (see Figure 18 for more detail data between 0 to 30 minutes). It shows that both of the results are skewed to the right with mostly less than 1.5 hours of queue wait time. Figure 19 shows the queue wait times on both machines on every trial.

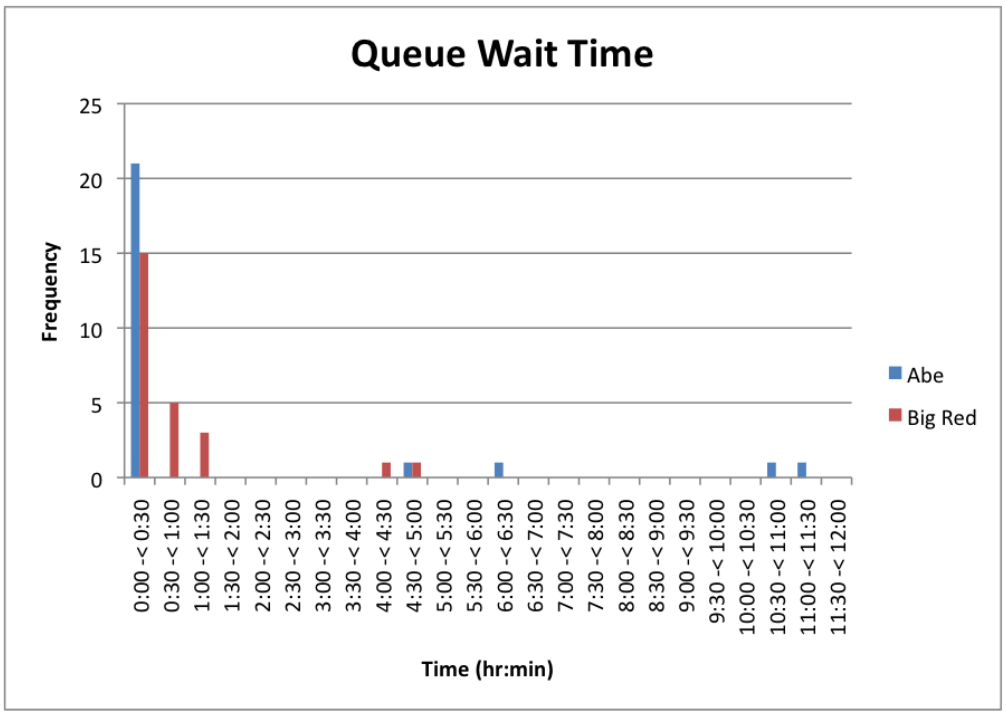

Figure 17: Histogram of Queue Wait Time

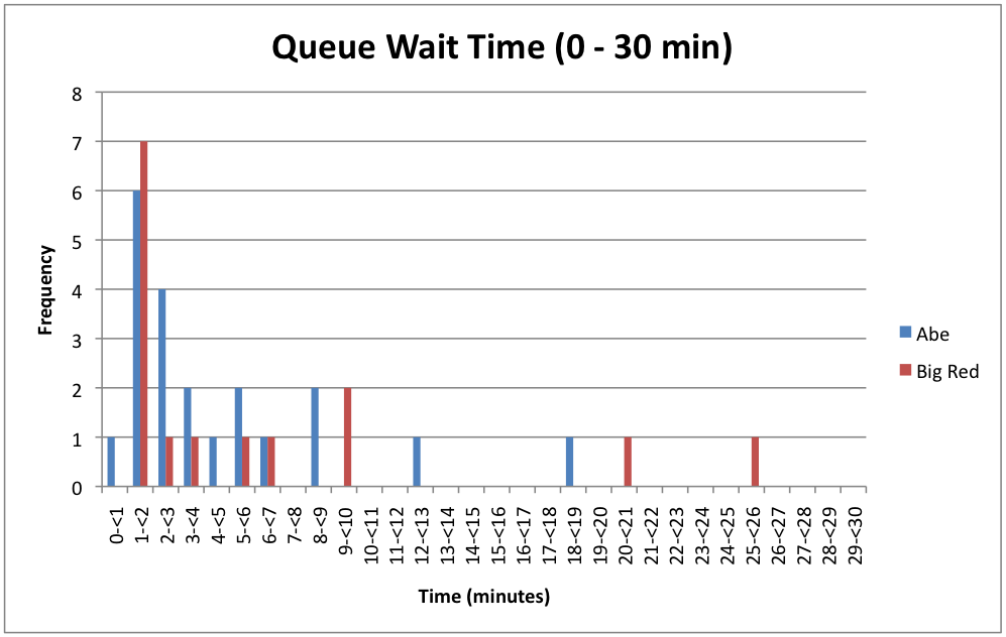

Figure 18: Histogram of Queue Wait Time (between 0 to 30 minutes)

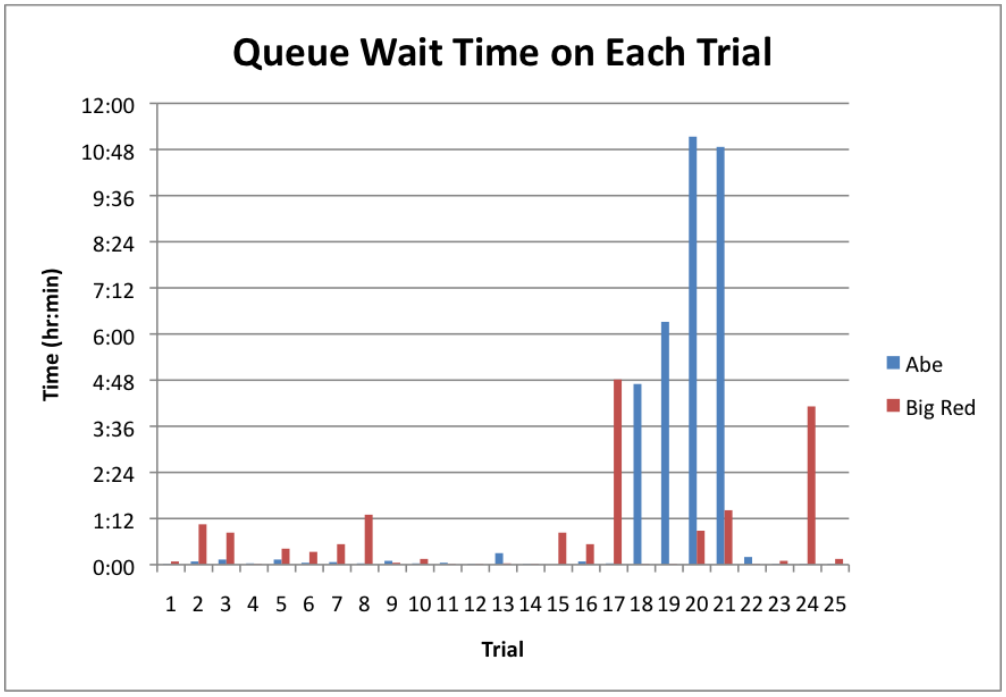

Figure 19: Queue Wait Time on Each Trial

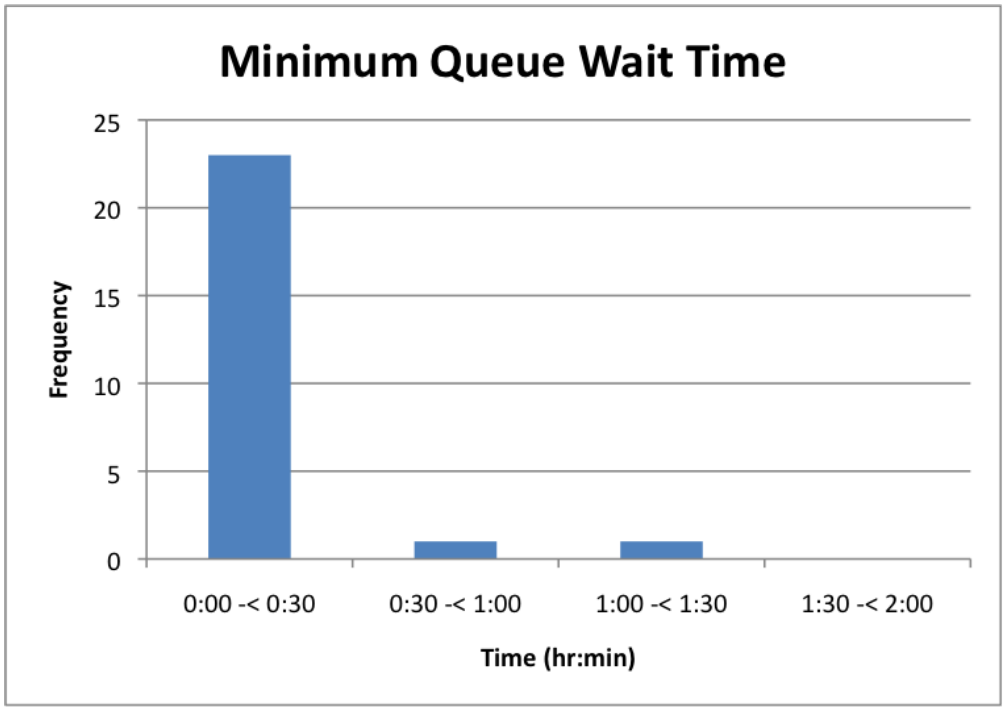

Figure 20: Histogram of Minimum Queue Wait Time

Table 8: Minimum Queue Wait Time (unit: minutes)

Average	7
Sample Variance	365
Standard Deviation	19

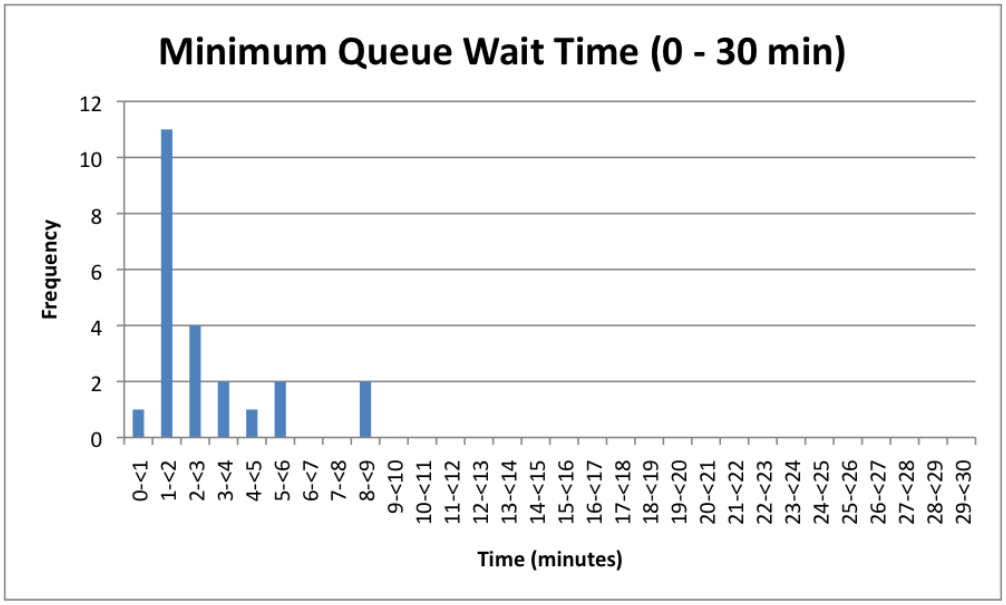

Figure 21: Histogram of Minimum Queue Wait Time (between 0 to 30 minutes)

Additionally, we obtained the statistical results of minumum queue wait time and difference between the maximum and minimum queue wait time from the experiment. Table 8 and Figure 20 show the average, sample variance, standard deviation, and the histogram of the minimum queue wait time (see Figure 21 for more detail data between 0 to 30 minutes). The minimum is the shortest queue wait time between Abe and Big Red on each trial. In the experiment, none of the minumum queue wait time is longer than 1.5 hours which occurred a number of times in the experiment result shown in Figure 17.

Figure 22 shows the histogram of the difference between the maximum and minimum queue wait times for each outcome (see Figure 23 for more detail data between 0 to 30 minutes).

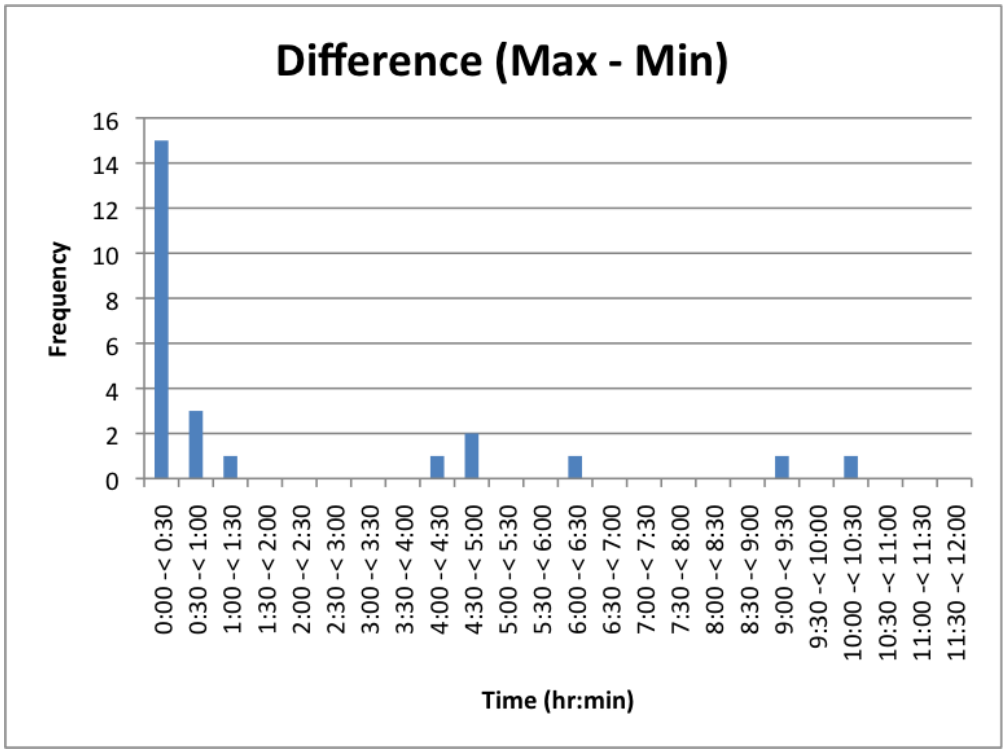

Figure 22: Histogram of Queue Wait Time Difference (Max - Min)

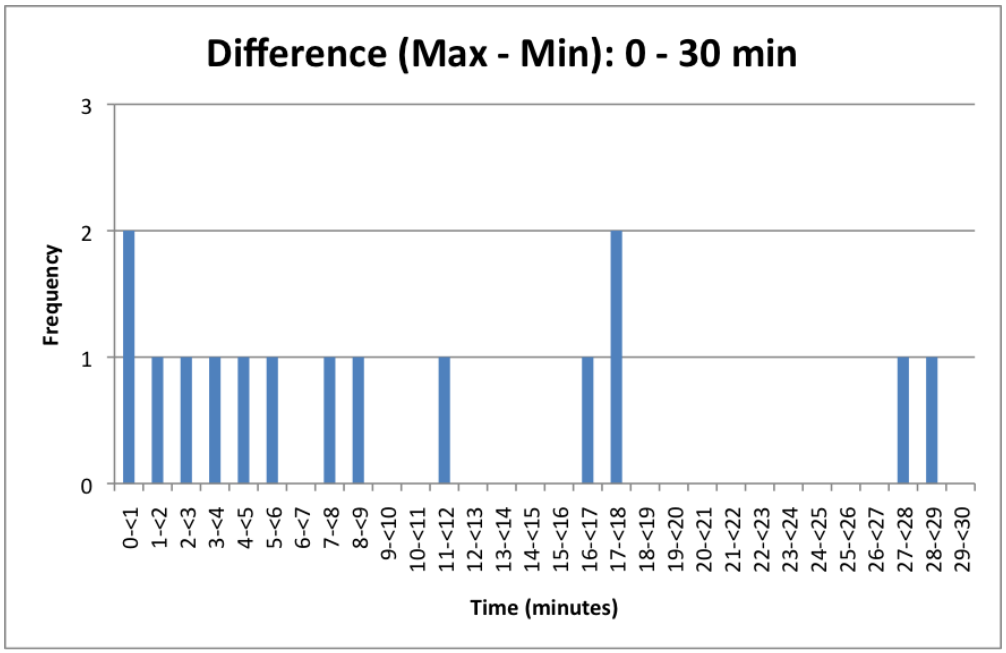

Figure 23: Histogram of Queue Wait Time Difference (between 0 to 30 minutes)

6.2.4 Conclusion

Although approximately 80% of the queue wait times are less than one hour as shown in Figure 17, a few outcomes have significantly long queue wait times. This may cause an unexpectedly long delay of the deployment (without dynamic workflow) with about 20% of probability, which is not desirable for the threat management.

However, in the dynamic workflow system, the minimum queue wait time among the resources can be the "actual" queue wait time for the application deployment. Since the minimum values cannot be greater than any of the outcome values, the minimum queue wait times always have the smallest values. Therefore, the queue wait time can be reduced in the dynamic workflow. In the experiment, about 90% of the minimum queue wait time was less than 10 minutes. This result shows that our framework can significantly shorten the queue wait time for the start of the deployment.

In addition, the different queue wait times on the different resources can be practically used in the workflow. These time differences can be used for the computations in our workflow rather than waiting, so that the "overall" run time can be reduced by the time difference.

6.3 Performance Comparison

We have compared the performance of the original application [2] to the performance of the application modified for the fault-tolerant workflow.

6.3.1 Objective

The mofication of the application for the fault-tolerant workflow may cause performance degradation of the original applcation. The goal of this experiment is to examine the overhead caused by the modification.

6.3.2 Test Setup

This experiment was conducted under the same condition as described in Section 6.1.2 except the number of generations and the resources used. Since there is a linear relationship between the run time and the number of generation, a set of number of generations (10, 20, 30, 40, and 50) is used instead of using every number of generation between 1 and 50. Also, only the Abe machine is used for this experiment since it has better compute performance than the one of the Big Red machine, so that we can use the resource usage hours efficiently. In addition, the generations to be computed are not divided for the comparison of the computational performance.

6.3.3 Result

Figure 24 shows the result of the comparison on the Abe machine. The run time difference in each of the results is equal to or smaller than 7 seconds, and the ratio of the run time increment gets closer to 1 as the number of generation increases.

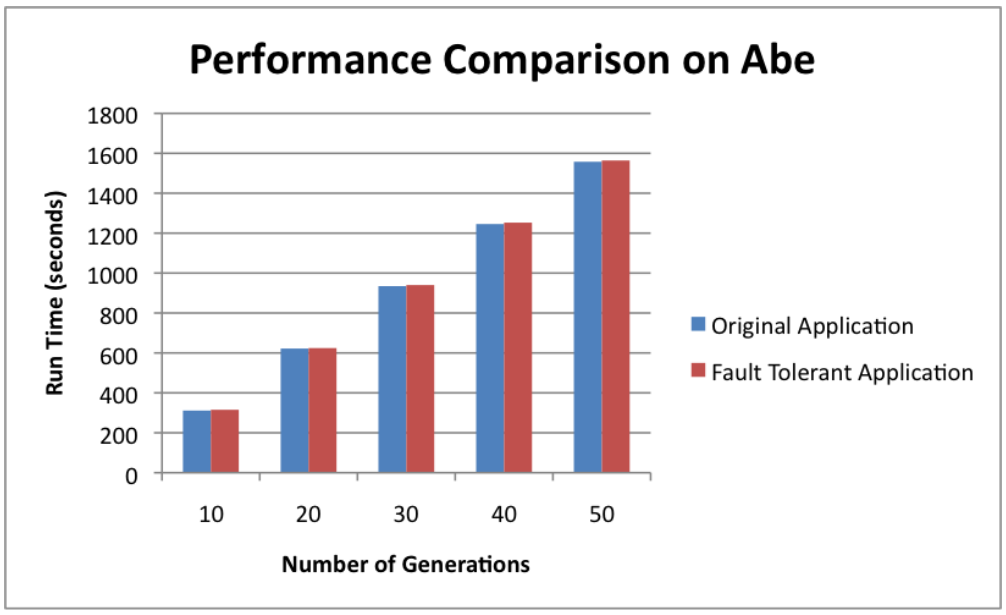

Figure 24: Performance Comparison between the original application and the fault-tolerant application on the Abe machine

6.3.4 Conclusion

The overhead caused by the modification is negligible.

6.4 Examination of Different Type of Deployment

We evaluated the total run time of each type of deployment: original, static, and dynamic workflow.

6.4.1 Objective

The purpose of this experiment is to examine how the fault-tolerant workflows (using the static job dependency and the dynamic job dependency strategy) behave in the real Grid environment with a simple workflow setup. Since the Grid environment changes dynamically, it is difficult to determine the best workflow strategy without information on how the dynamic Grid environment affects the workflow deployment. Thus, we use a simple workflow model to investigate how the dynamic workflow can adapt to the dynamic environment. We also compare the fault-tolerant deployments with the original deployment.

6.4.2 Test Setup

Since there are some limitations to evaluate the performance including queue wait time, which are the fact that the queue wait time may be very different from time to time and the difficultness to make the statistics for the total run time due to the limited resource usage hours, we evaluate several different cases of the deployments of the original application and fault-tolerant application including queue wait time under the similar condition; the queue wait time for the first submitted jobs for the different kinds of deployments within each site is the same or very similar, i.e., less than 3 minutes of time difference, so that each of the deployments can be compared directly to each other. Also, the experiment results that do not meet the condition are not included in this evaluation.

In order to make the similar condition of queue wait time for the comparison of the different

workflow strategies of the fault-tolerant application and the original application, we submit each of the jobs to run on a single node, so that there can be more chances that the same number of available nodes can be allocated at the same time for more number of jobs.

For this experiment, the Abe and Big Red machines are used for the different type of deployments. The number of cores used for a job on the Abe machine is 8, and the number of cores used for a job on the Big Red machine is 4. Each job on each machine is scheduled to run on a single node, and the number of generations for each type of deployment is 20.

For the comparison, we compare the total run time (queue wait time plus run time of the job) between the original application and the fault-tolerant application with the distribution of the generations on the two different sites, Abe and Big Red. In addition, we compare the two different workflow strategies, non-dynamic and dynamic job dependency within this comparison. Table 9 summarizes each deployment for the experiments. All the jobs are submitted at the same time except the second job in the non-dynamic job dependency strategy. As shown in the table, in the static job dependency deployment, the job on Big Red computes the generation from 11 to 20 after the computation of generation from 1 to 10 on Abe, and the job on Big Red is submitted after the finish of the job on Abe.

In the dynamic workflow, how to distribute the generations among the different compute performance machines needs to be considered to reduce the total run time. However, assigning more number of generations to a better compute performance machine does not always guarantee a smaller run time since the workflow deployment is affected by the queue wait time on each job. If a better compute performance machine has a much longer queue wait time than the one of the worse compute performance machine, assigning a larger number of generations to the job scheduled to run on the worse compute performance machine might cause a smaller run time. As the purpose of this experiment, we divide the generations equally among the different sites for a simple workflow model.

Table 9: Detail of Water Threat Management Application Deployment

Version	Workflow	Site Name	Number of Jobs	Generation Range
Original	-	Abe	1	1-20
Original	-	Big Red	1	1-20
Fault-tolerant	static	Abe, Big Red	2	1-10 (Abe), 11-20 (Big Red)
Fault-tolerant	dynamic	Abe, Big Red	2	1-10, 11-20

6.4.3 Result

We conducted five experiments; however, we do not include two of them since the two experiment results do not meet the condition as described in Section 6.4.2, which is that the first jobs within the same site should have similar queue wait times, so that the different type of deployments cannot be compared fairly in the two experiment results.

Figure 25 shows the case that the jobs on Big Red started earlier than the jobs on Abe. In the graph, the job run time includes the time for waiting for the inputs from the preceding job, from the job on Big Red in this case, thus the run time of the job in the dynamic deployment is shown to be longer than the actual compute time. Figure 26 shows the comparison of the total run time of each deployment. Although the generations are distributed equally on Abe and Big Red, the total run time in the dynamic job dependency strategy is different from the one in the non-dynamic job dependency strategy. In the non-dynamic strategy, the job on Big Red for the generation range from 11 to 20 was submitted after the finish of the job on Abe in the workflow while the generation range from 1 to 10 was assigned to the job on Big Red in the dynamic workflow, so the total run time is longer than the dynamic strategy.

In the second case, shown in Figure 27, the jobs on Big Red started earlier but with the similar queue wait time on each site. Like the workflow of the non-dynamic job dependency, the workflow of the dynamic job dependency has the same job "site" order of the generation range as the one of the non-dynamic job dependency. However, the second job in the non-dynamic workflow was submitted after the job on Abe was done, and the queue wait time of the second job was longer compared to the others, so the dynamic workflow shows shorter total run time than the total run time of the non-dynamic workflow as shown in Figure 28.

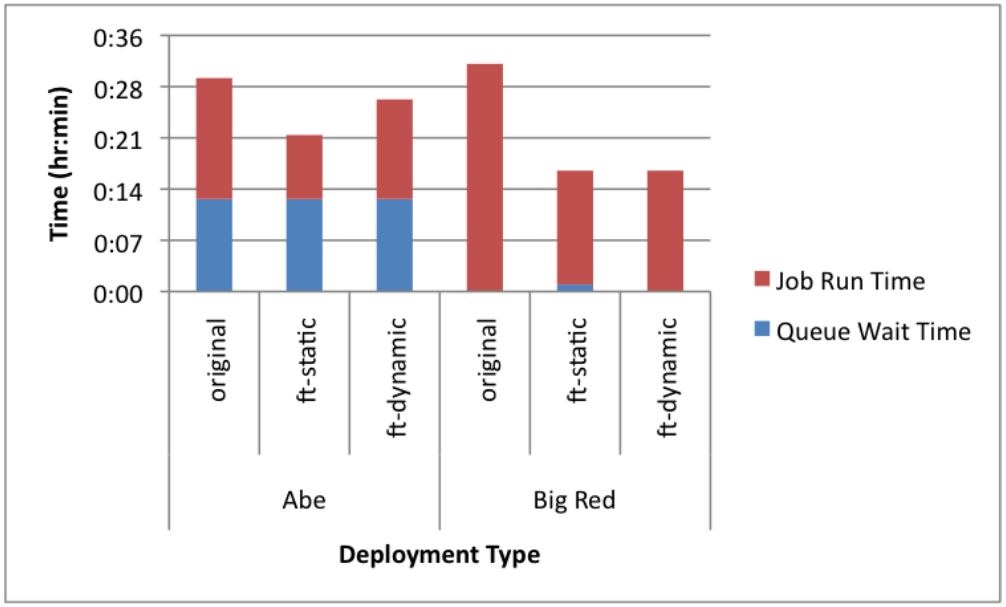

Figure 25: Queue Wait Time and Job Run Time of the original application and the fault-tolerant application with Static/Dynamic Workflow: Case 1

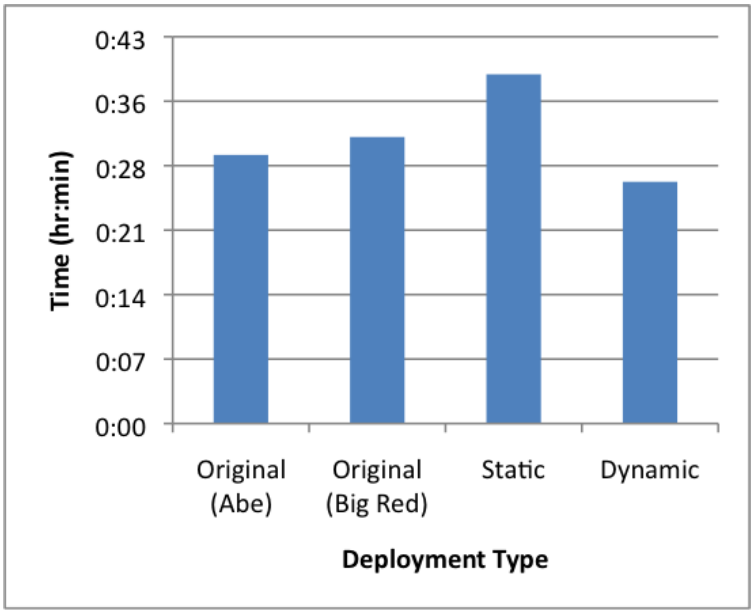

Figure 26: Total Run Time Comparison between the original application and the fault-tolerant application with Static/Dynamic Workflow: Case 1

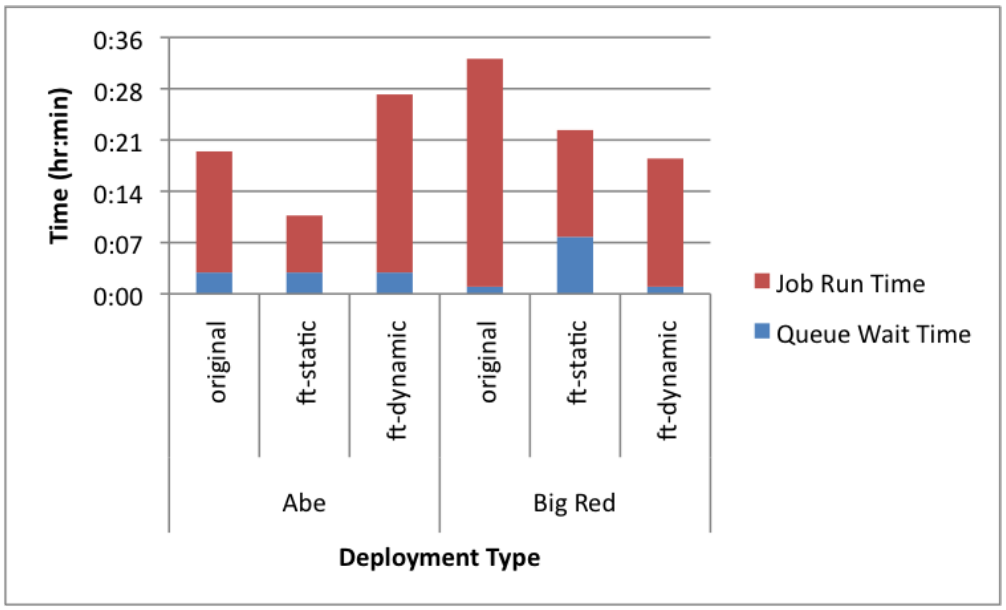

Figure 27: Queue Wait Time and Job Run Time of the original application and the fault-tolerant application with Static/Dynamic Workflow: Case 2

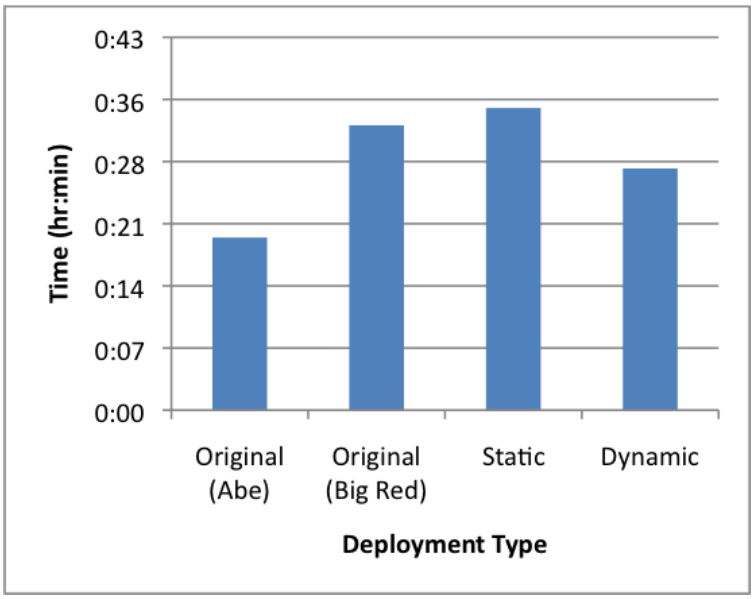

Figure 28: Total Run Time Comparison between the original application and the fault-tolerant application with Static/Dynamic Workflow: Case 2

Contrary to the second case of the fault-tolerant workflows, in the case shown in Figure 29, the total run times of the dynamic and non-dynamic workflow are very similar. Although the jobs on Abe started earlier, the second job in the non-dynamic workflow had very short queue wait time, so that the second jobs of both strategies could finish at very similar time. This implies that the total run time of the dynamic workflow is not always smaller than the one of the non-dynamic workflow in the case that the job "site" order is the same between the dynamic and non-dynamic workflow. However, it should be considered that each of the deployments is not in exactly the same environment in this experiment as described in the previous section. It should be noted that this experiment shows the possible result of each type of deployment.

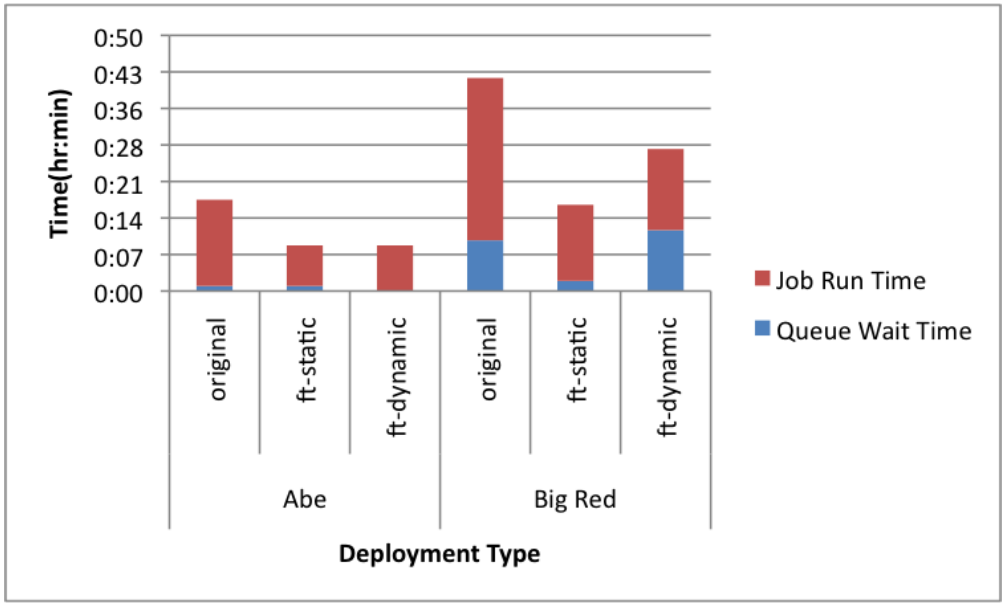

Figure 29: Queue Wait Time and Job Run Time of the original application and the fault-tolerant application with Static/Dynamic Workflow: Case 3

In addition, the three cases indicate that the total run time of the original application deployment may be longer or shorter than the total run time of the fault-tolerant application deployments based on the different queue wait time.

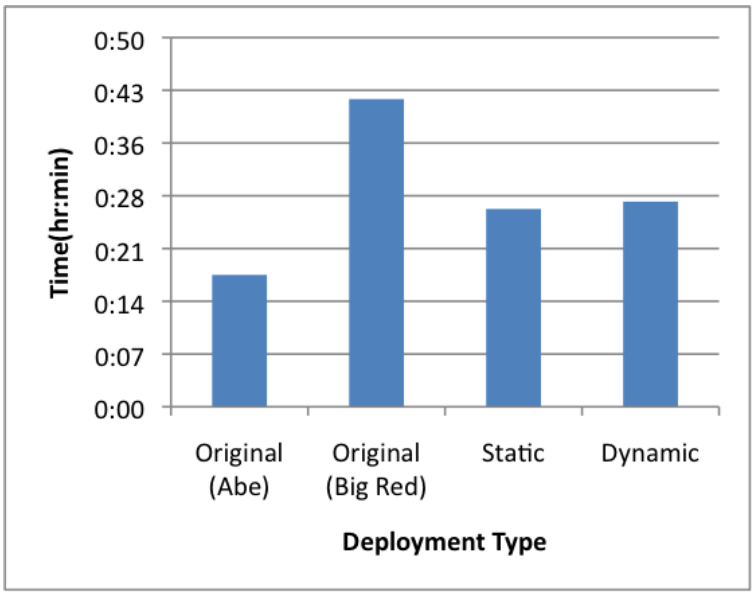

Figure 30: Total Run Time Comparison between the original application and the fault-tolerant application with Static/Dynamic Workflow: Case 3

6.4.4 Conclusion

As shown in Figure 26, the total run time can be reduced in the dynamic fault-tolerant workflow compared to the original one. The deployment with dynamic job dependency reduced the total run time around 10% compared to the total run time of the original deployment on Abe and around 16% compared to the one on Big Red in this experiment. However, the dynamic workflow strategy may cause an "idle" processing time in the job execution depending on the job start time.

6.5 Performance Estimation

In this section, we obtain the average run time of the dynamic workflow and the original workflow using the performance model.

6.5.1 Objective

The purpose of this estimation is to evaluate the average run time of the dynamic workflow deployment with the fault-tolerant queuing framework including failures, and we compare the performance between the dynamic and original workflow.

6.5.2 Test Setup

We develop a performance model based on the equations in Section 5. The following is the performance model description for each deployment type.

The total run time of a job is the job's run time including its queue wait time. Based on equation (8), we define the total run time of the original deployment as

$$T(n)_{original} = (\frac{1}{(1-p)^t} + 1) \cdot \frac{t}{2} + \frac{1}{(1-p)^t} \cdot q \tag{12}$$

where t is the run time of n generations on a specific machine, p is the failure probability on a specific machine, and q is the average queue wait time on a specific machine. Note that $\frac{1}{(1-p)^t}$ is derived from the expected number of trials to run a job until it succeeds (equation (5)).

We use the values for each variable from the outage analysis in Section 2, the performance data from Section 6.1, and the queue wait time experiment results in Section 6.2.

For the total run time of the dynamic workflow deployment, we define the model based on equation (3) as

$$T_{dynamic}(k) = \begin{cases} \frac{1}{(1-p_1)^{t_1}} \cdot q_1 + (\frac{1}{(1-p_1)^{t_1}} + 1) \cdot \frac{t_1}{2} & k = 1 \\ \max(q_k, T_{dynamic}(k-1)) + (\frac{1}{(1-p_k)^{t_k}} + 1) \cdot \frac{t_k}{2} & k > 1 \end{cases} \tag{13}$$

where k is the number of jobs in the workflow, q_i is the ith shortest average queue wait time ($1 \leq i \leq k$) among the different machines, and p_i and t_i are the failure probability and the run time of a job, respectively, corresponding to q_i.

With this model, we evaluated the run times of 10, 20, 30, 40, 50, 100, 150, and 200 generations of the original and dynamic workflows with the statistical data shown in Table 10. Also, the number of cores and nodes for each machine is the same as in the experiment in Section 6.1 as this statistical data is obtained from the experimental result.

Table 10: Statistical Data for Performance Estimation

	Abe	**Big Red**
Average Queue Wait Time (min)	82	42
Failure Rate per Year	0.065	0.050
Runtime per Generation (min)	0.52	2.07

6.5.3 Result

First, we analyze the run time affected by the different generation distribution. As the generations are distributed in the dynamic workflow, the different distributions cause the different run time. Figure 31 shows the change of run time based on the different generation distributions. (The original workflow on each machine has no generation distribution.) Based on the data in Table 10, a job on Big Red always starts first since it has a shorter average queue wait time (42 minutes) than the one on Abe (82 minutes). Therefore, the first generations (starting from generation 1) are computed on the Big Red machine. When most generations are computed on the Abe machine, the run time is almost the same as the original workflow run time on Abe. However, as more generations are computed on the Big Red machine, the run time decreases. Note that there are 40 minutes of difference between the two queue wait times. More generations can be computed on Big Red before the job on Abe starts. In addition, if the run time of the number of generations computed on Big Red exceeds the queue wait time on Abe, the run time increases since the job on Abe should wait until the job on Big Red finishes its computation.

However, it should also be considered that if the queue wait time on Abe is shorter than the one on Big Red, the dynamic workflow cannot have better performance than the performance of the original deployment on Abe. Despite this fact, this experiment still shows valuable information in case that a better compute performance machine has a longer queue wait time than the queue wait time of a worse compute performance machine. As shown in Figure 19 in Section 6.2, 16% of the results is the cases in which a better compute machine has significantly longer queue wait times than the queue wait times on the worse compute performance machine.

The most efficient distribution of the generations in this estimation is dividing generation from 1 to 19 (Big Red) and from 20 to 50 (Abe) for 50 generations. Also, the other results show that

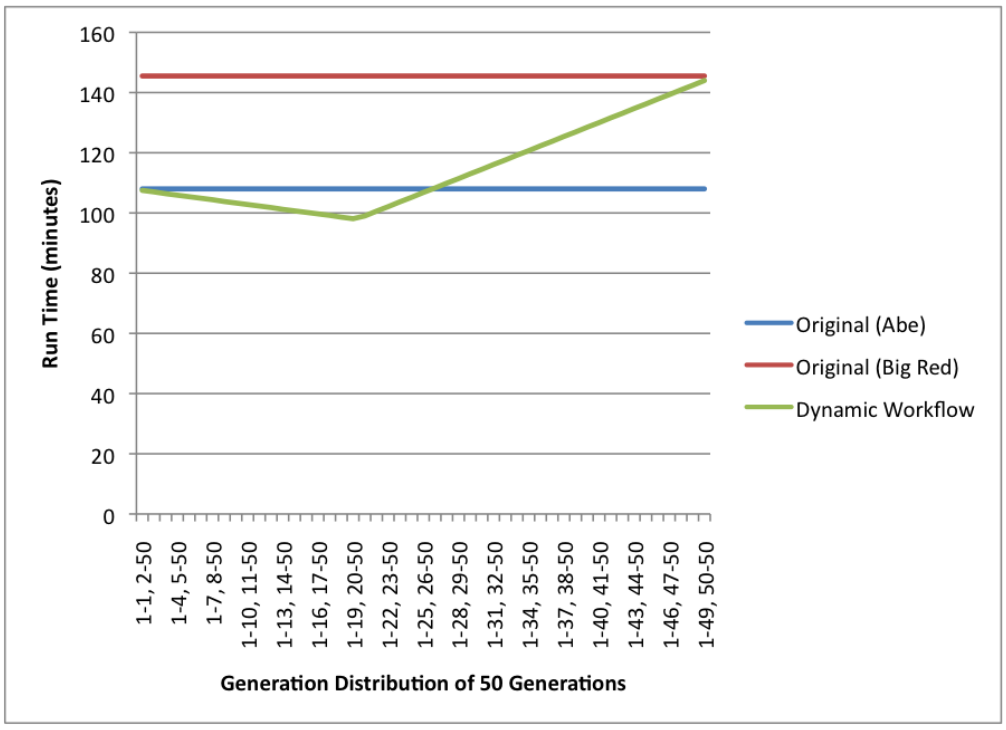

Figure 31: Run Time of Different Generation Distribution (50 Generations)

generation distribution from 1 to 19 on Big Red and the rest on Abe has the shortest run time for 100, 150, and 200 generations.

Figure 32 shows the estimation results of comparison between the run time of the original deployment on each machine and the run time of the dynamic workflow based on the different number of generations. The run time of the dynamic workflow shown in the graph is the shortest run time among the different generation distributions for each number of generations.

The dynamic workflow is effective if the number of generations is equal to or greater than 20. Additionally, the run time of the dynamic workflow is the same if the number of generations is smaller than 20 since the last generation needs to be computed on Abe after waiting for the start of the job on Abe. This result shows that the dynamic workflow may not be efficient for a small number of generations.

In this estimation, the run time of the dynamic workflow is approximately 10 minutes shorter than the original deployment on Abe if the number of generation is equal to or greater than 20. This 10 minutes of time is the amount of time reduced by the queue wait time difference between

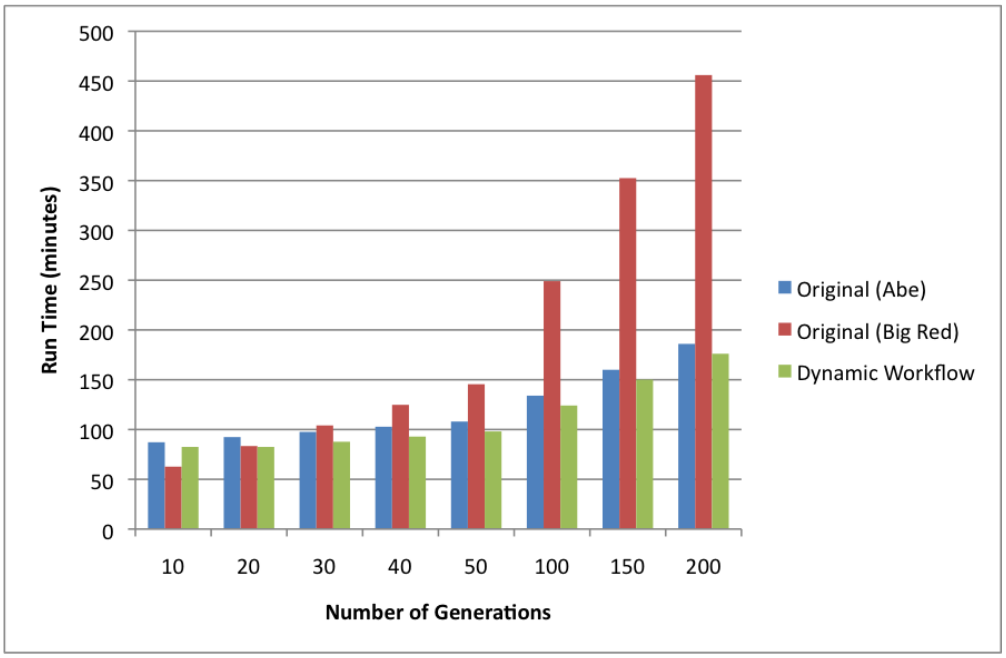

Figure 32: Run Time Comparision

the two machines. While one job is waiting for its start, another job already started on the other machine computes the generations.

6.5.4 Conclusion

Depending on the queue wait time and the number of generations, the dynamic workflow can have better performance than the original deployment. Although the dynamic workflow may not result in best performance in every case, the estimation shows that, on average (note that the average value, i.e., queue wait time, is applied in the estimation), the dynamic workflow has a shorter run time than the original deployments if the number of generations is large.

6.6 Worst Case Run Time

In this section, we describe the worst case simulation of our dynamic workflow and the original workflow.

6.6.1 Objective

The objective of this experiment of simulation is to compare the worst case of performance between the dynamic workflow deployment and the original deployment. Since the threat management system should deliver the results in a timely manner in any circumstances even though the failure rate is low, it is very important to observe the worst case of performance of the dynamic workflow and how much the dynamic strategy can increase the performance compared to the original deployment.

6.6.2 Test Setup

For the worst case run time comparison, we simulate the dynamic workflow and original deployment using the three different machines (Abe, Big Red, and Queen Bee) based on the outage data described in Section 2. We use the performance data in Section 6.1 and the queue wait time data in Section 6.2 for Abe and Big Red. For Queen Bee, we use the performance and queue wait time data obtained from our experiment with the different requirement of the number of cores used (see Table 11) from the one for Abe and Big Red. (Queen Bee allows only 8 cores for MPI [22] jobs.) For the queue wait time setup, we use the queue wait time outcomes (Table 12) that we obtained from our experiment.

Table 11: Simulation Setup

	Abe	**Big Red**	**Queen Bee**
Run Time per Generation (min)	0.52	2.07	1.02
Number of Cores	16	16	8

The worst case of run time is obtained as follows. We simulate submitting jobs with each deployment every 5 minutes from January first of 2009, 00:00 EST throughout the year of 2009. The 25 trials are mapped to every deployment. For example, a job submission at time January 1st 2009, 00:00 EST is mapped to all the 25 queue wait times, so that there are 25 results for each deployment. From the simulation, the longest run time for each number of generations is selected as the worst case of run time. In addition, in case of the generation distribution, the generations are

Table 12: Queue Wait Time Setup (unit: min)

Trial #	Abe	Big Red	Queen Bee
1	1	5	99
2	5	63	0
3	8	50	258
4	2	1	26
5	8	25	4
6	3	20	0
7	4	32	18
8	2	78	0
9	6	3	1
10	2	9	302
11	3	1	0
12	1	1	0
13	18	2	60
14	1	1	5
15	0	50	0
16	5	32	0
17	2	289	50
18	282	1	4
19	379	1	267
20	668	53	0
21	652	85	50
22	12	1	16
23	1	6	56
24	1	247	24
25	1	9	0

equally distributed among the different jobs.

The dynamic workflow simulation algorithm is described in Algorithm 2. The original deployment simulation can be done with Algorithm 2 if only one machine is used and there is no generation distribution.

6.6.3 Result

Figure 33 shows the run time of each deployment. The numbers in the parentheses indicate the number of jobs in the workflow. For example, if the number is 2, this means that the generations are divided into two pars as two different jobs. In addition, the machine names in the parentheses

Algorithm 2 Water Threat Management Dynamic Workflow Simulation Algorithm

```
distribute generations
while(until all generations are computed){
    calculate job submission time for each machine based on the resource outage time.
    calculate job start time based on the queue wait time and resource outage time
    sort the job start time on each machine
    for each job with the ascending order of the job start time {
        run job
        job end time ← job start time + job run time
        if (job fails)
            if (job fail time > the next job start time)
                the next job start time ← job fail time
            reschedule the failed set of generations
        else
            if (job end time > the next job start time)
                the next job start time ← job end time
    }
}
workflow run time ← job end time − the original job submission time
```

indicate that those machines are used for the deployment.

In general, the worst case run time of the original deployments show somewhat exponential growth as the number of generations increases (Figure 35). This indicates that a job with a longer run time has higher chances of failure than a job with a smaller run time, so that a job with a longer run time likely needs to restart more often.

Figure 36 shows the run time of each dynamic workflow deployment. Generally, distributing generations into more number of jobs results in better performance (see Figure 37, 38, 39, and 40). If the generations are distributed, each of the jobs has a smaller run time, so that there are lower chances of failure and the amount of calculation can be reduced since only the subset of generations need to be processed again in case of failure. In addition, when multiple jobs are launched in the workflow and if one job fails, the other job may start immediately since all the jobs in the workflow are submitted at the same time (the dynamic job dependency strategy).

As shown in Figure 34, some of the dynamic workflow deployments have worse performance than the performance of the original deployment on Abe or Queen Bee. This is caused by allocat-

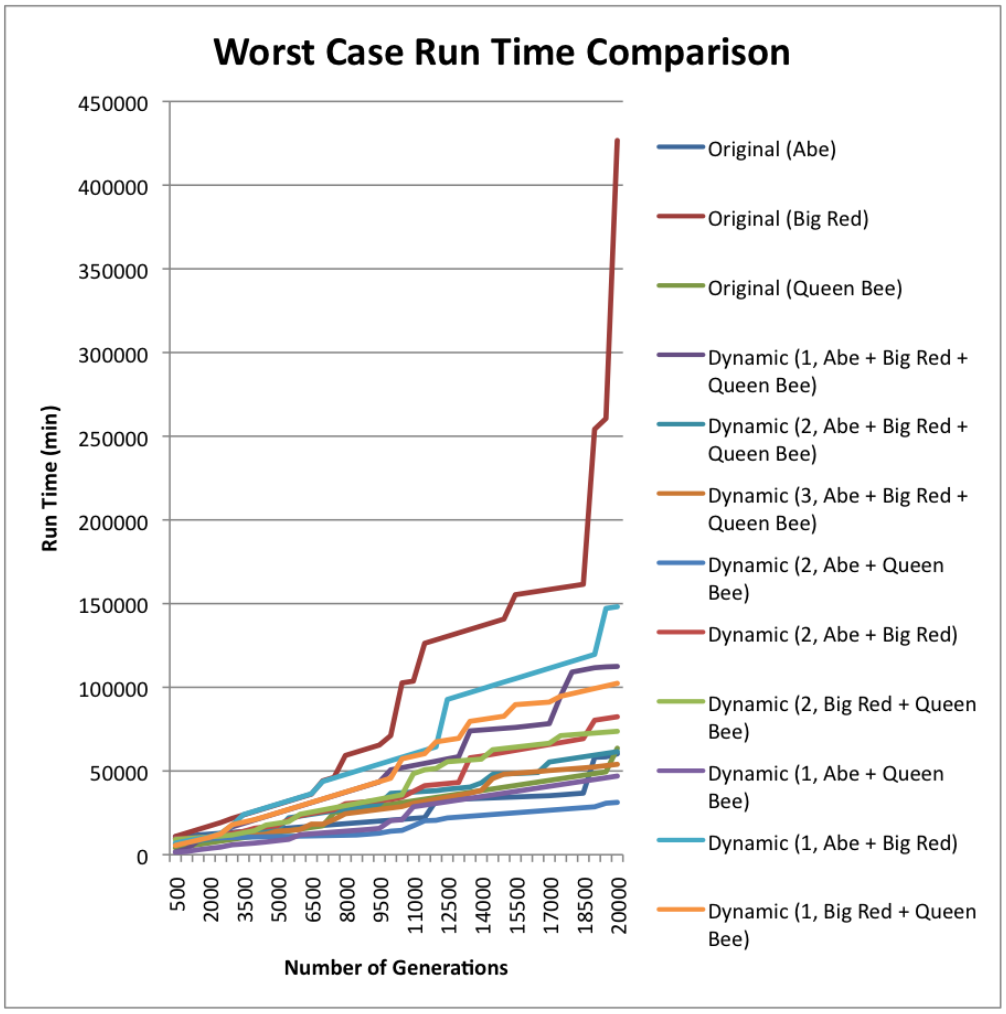

Figure 33: Worst Case Run Time Comparison

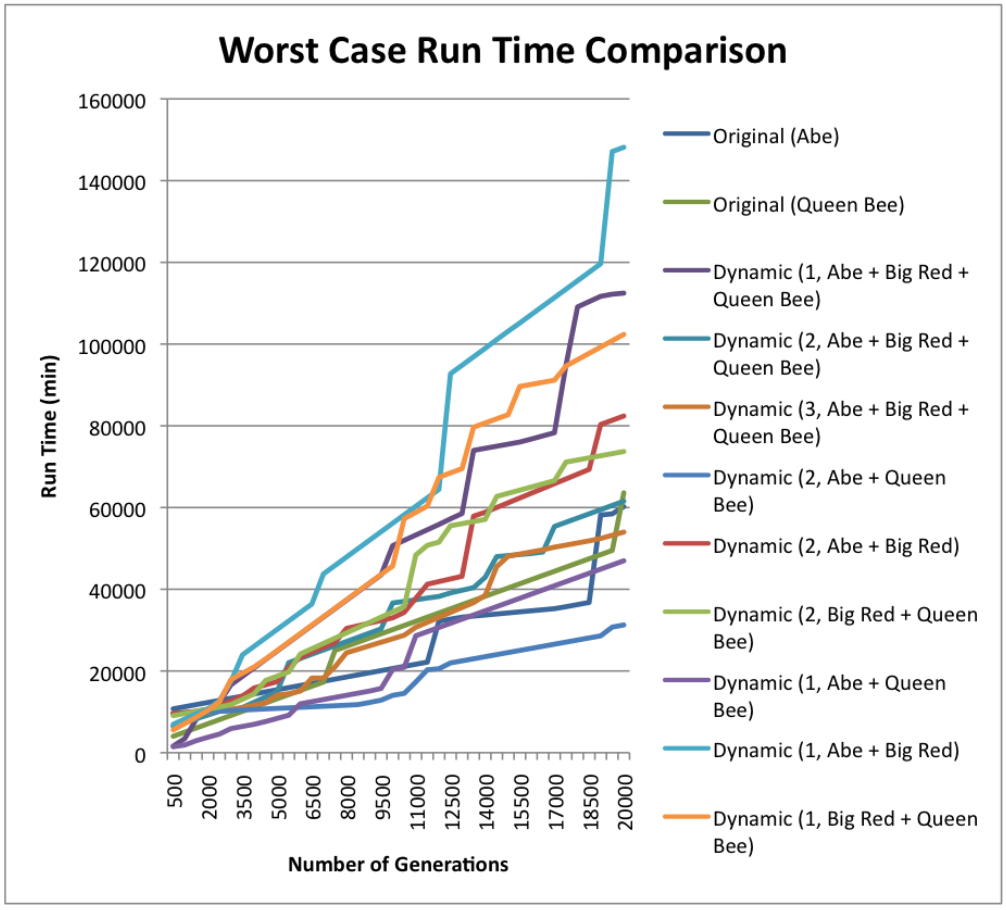

Figure 34: Worst Case Run Time Comparison except the original deployment on Big Red

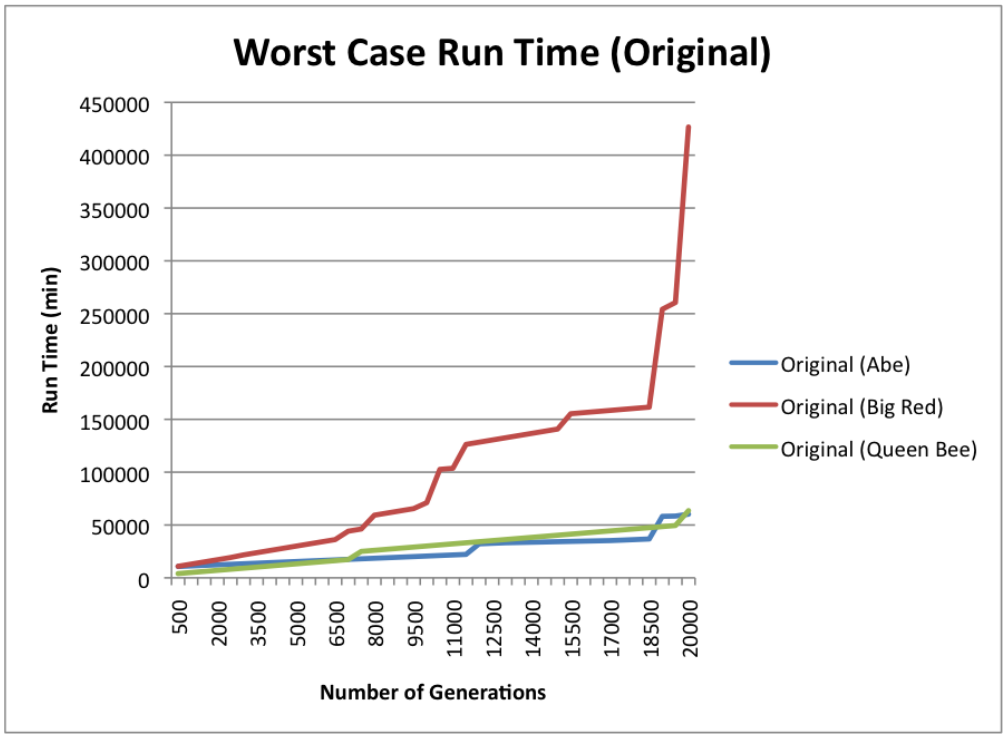

Figure 35: Worst Case Run Time Comparison: Original Deployment

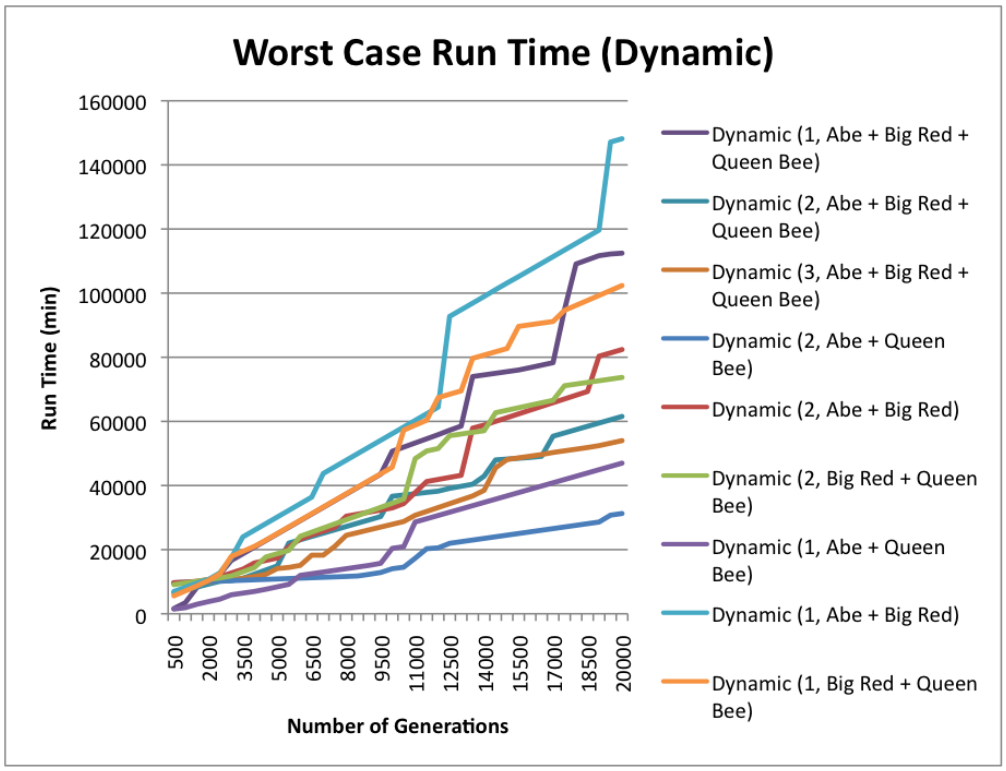

Figure 36: Worst Case Run Time Comparison: Dynamic Workflow Deployment

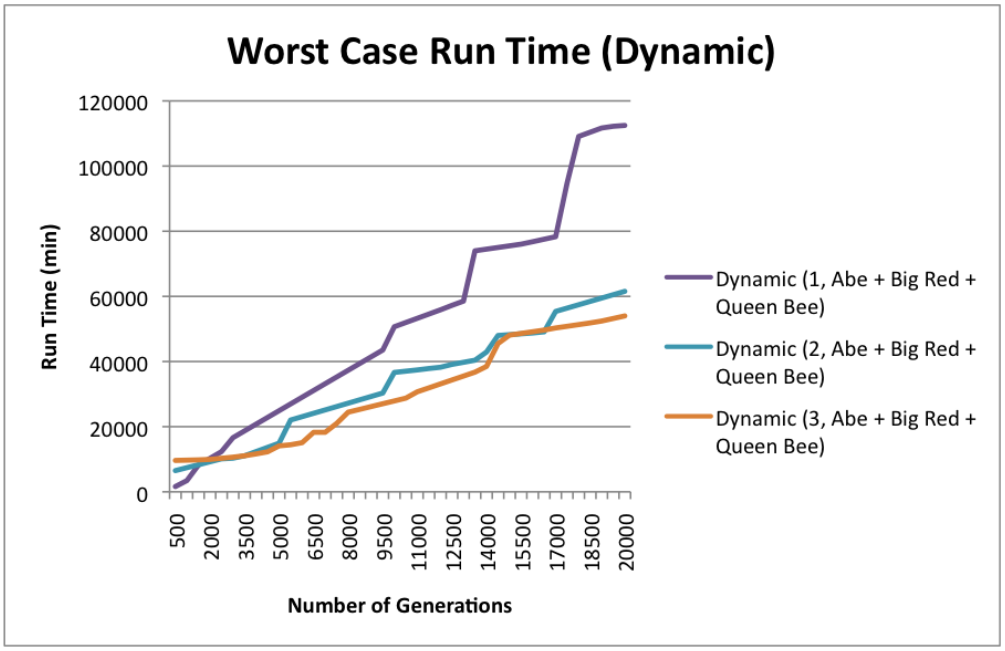

Figure 37: Worst Case Run Time Comparison with different number of jobs on Abe, Big Red, and Queen Bee

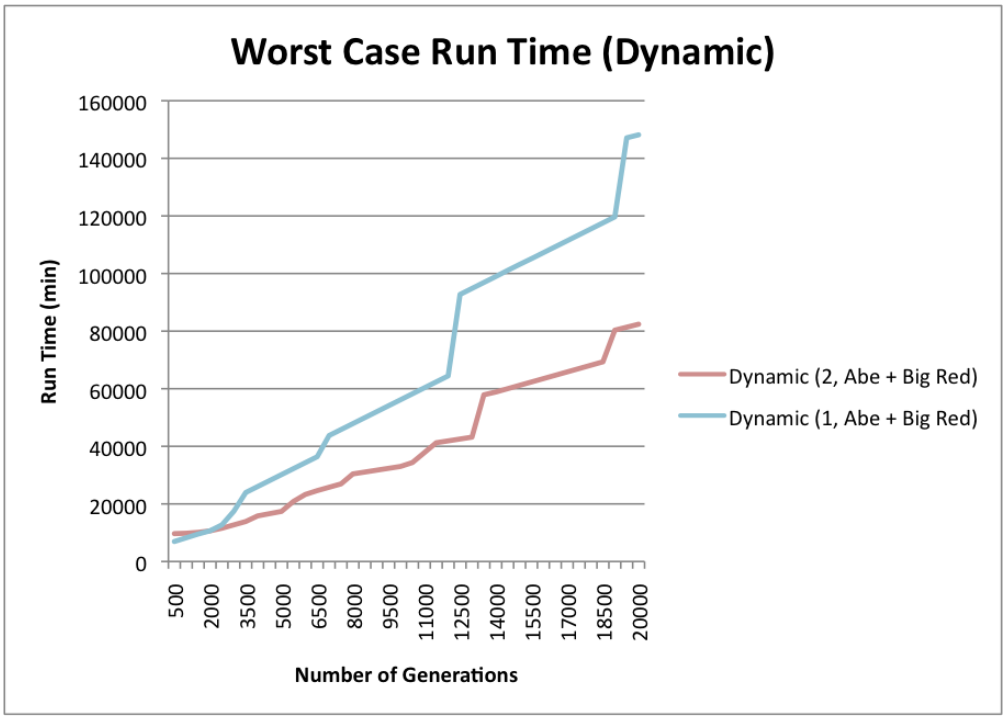

Figure 38: Worst Case Run Time Comparison with different number of jobs on Abe and Big Red

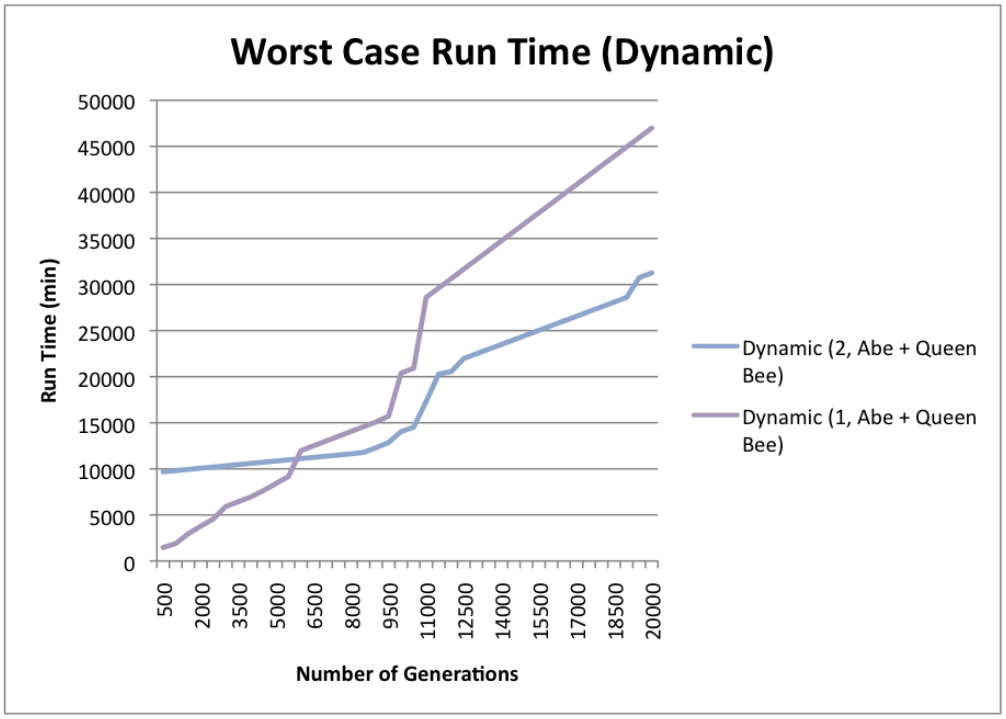

Figure 39: Worst Case Run Time Comparison with different number of jobs on Abe and Queen Bee

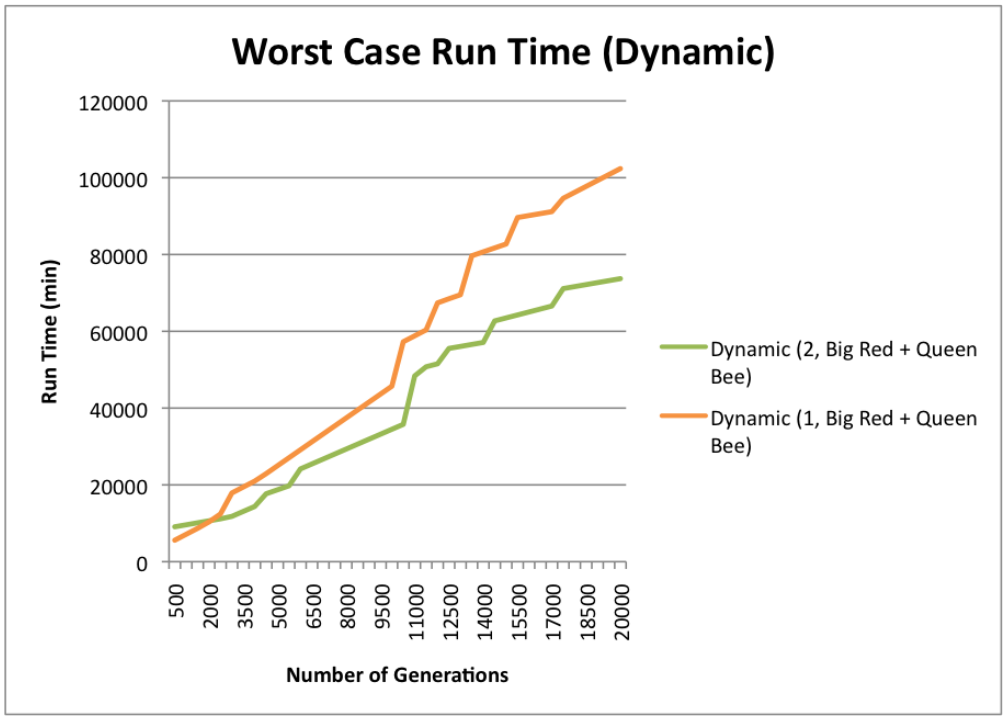

Figure 40: Worst Case Run Time Comparison with different number of jobs on Big Red and Queen Bee

ing the Big Red machine, which has much worse compute performance compared with the other machines, for the dynamic workflow. Figure 41 clearly shows the effect of the Big Red machine. The result without using the Big Red machine has much better performance.

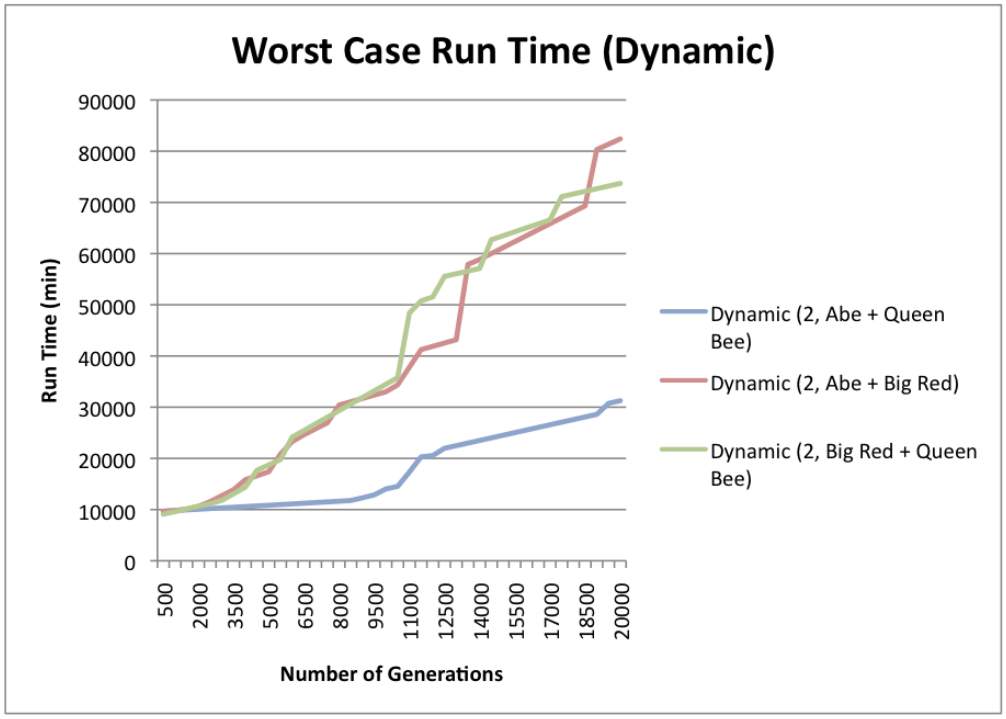

Figure 41: Worst Case Run Time Comparison with different combination of machines

The dynamic workflow deployment with two jobs on the Abe and Queen Bee machines shows the shortest run time among all the different deployments. When the number of generations is greater than a certain value as shown in Figure 42, the dynamic workflow always has better performance than that of the original deployments, and the improvement is significant.

6.6.4 Conclusion

With selecting good compute performance machines, the dynamic workflow strategy can significantly reduce the run time in the worst case. The improvement in the worst case is very critical in the Water Threat Management system [1] [2] since it deals with urgent events related to human safty. The system should deliver the services in real-time in any cases of the outages.

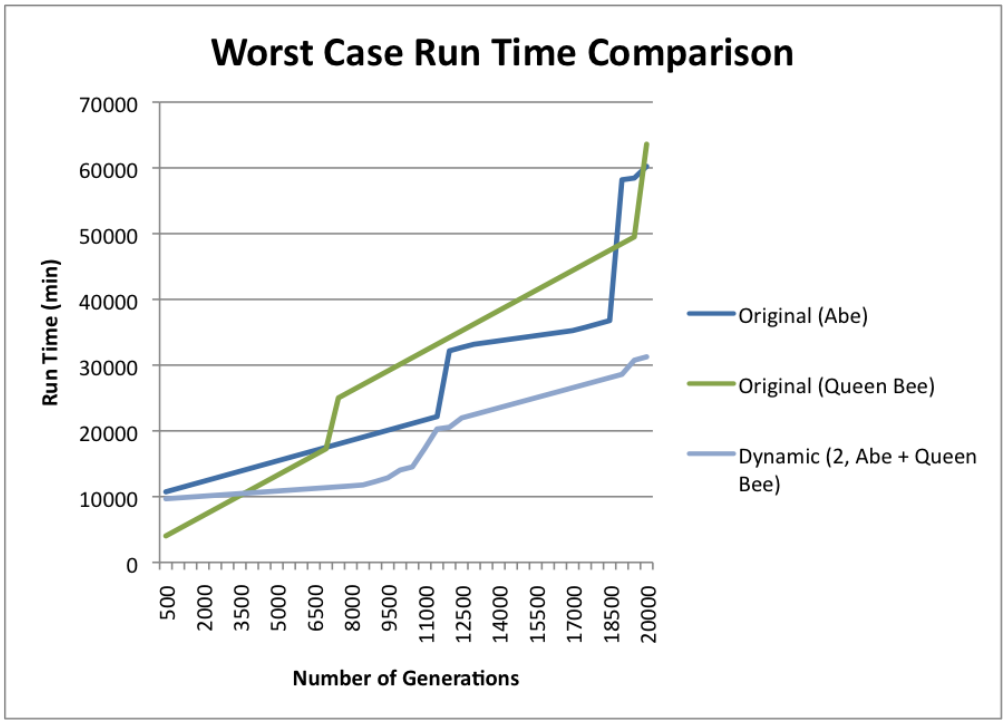

Figure 42: Worst Case Run Time Comparison

6.7 Five Number Summary for Run Time Simulation

Although we have examined the worst case run time of each deployment, the "general" performance of the dynamic and original deployment should also be considered. In this section, we describe the five number summary for the run time observations from our simulation.

6.7.1 Objective

This simulation is to examine and compare the "distribution" of the run time of each deployment by the five number summary, so that we can understand the "normal" case of performance.

6.7.2 Test Setup

The simulation setup is identical to the setup in Section 6.6.2, and like the setup in Section 6.6.2, we use Algorithm 2.

6.7.3 Result

Table 13 describes the five number summary for the run time of the original deployment on Abe and the dynamic workflow on Abe and Queen Bee. We select the result of the run time on Abe for the original deployment and the result of the run time on Abe and Queen Bee for the dynamic deployment for the comparison since each of those has the best performance for each type of deployment.

As shown in Figure 43, the median run time of the dynamic workflow (with two jobs for the generation distribution) is approximately 50% longer than the median run time of the original deployment on Abe since the half of the generations in the dynamic workflow are computed on Queen Bee, which has 50% slower compute performance than Abe's (see Table 11). However, the maximum run time (worst case run time) of the dynamic workflow is much shorter than the maximum run time of the original deployment as shown in Section 6.6.

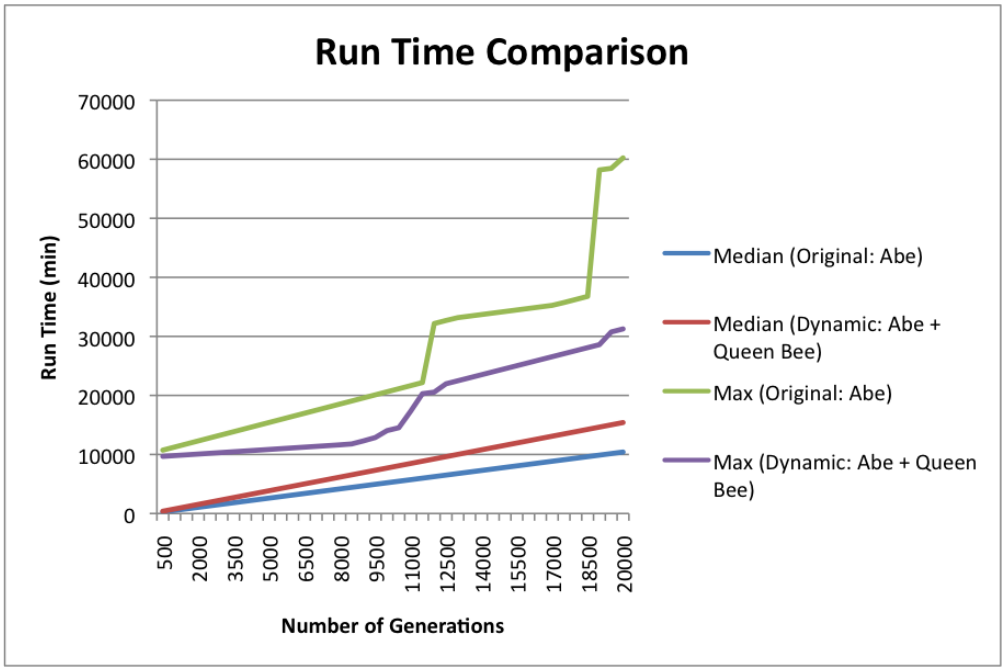

Figure 43: Median and Maximum Run Time Comparison

6.7.4 Conclusion

In general, the dynamic workflow has worse performance than the performance of the original deployment although significant improvement in the worst case performance is achieved. The main reasons for this result are that the low failure rate (approximately 5%) of the machines and the difference of compute performance between each machine. Without failure, the jobs do not need to restart, so that the run time is not increased by failure. In addition, since the generations are distributed, some of the generations are simulated on the worse compute performance machine. This is another main factor of the worse performance of the dynamic workflow in general case.

Table 13: Run Time Comparison: Five-Number Summary: unit (min)

	Original (Abe)					Dynamic (Abe + Queen Bee)				
#Gen.	**Min**	**Q1**	**Median**	**Q3**	**Max**	**Min**	**Q1**	**Median**	**Q3**	**Max**
500	260	261	264	278	10733	260	385	387	513	9678
1000	520	521	524	538	11253	520	770	772	788	9808
1500	780	781	784	798	11773	780	1155	1156	1163	9938
2000	1040	1041	1044	1058	12293	1040	1540	1541	1548	10068
2500	1300	1301	1304	1318	12813	1300	1925	1926	1933	10198
3000	1560	1561	1564	1578	13333	1560	2310	2311	2322	10328
3500	1820	1822	1825	1926	13853	1820	2695	2697	2707	10458
4000	2080	2082	2085	2313	14373	2080	3080	3082	3092	10588
4500	2340	2342	2345	2622	14893	2340	3465	3467	3477	10718
5000	2600	2602	2605	2882	15413	2600	3850	3852	3862	10848
5500	2860	2862	2865	3142	15933	2860	4235	4237	4247	10978
6000	3120	3122	3125	3402	16453	3120	4620	4622	4638	11108
6500	3380	3382	3385	3362	16973	3380	5005	5007	5023	11238
7000	3640	3642	3645	3930	17493	3640	5390	5392	5408	11368
7500	3900	3902	3905	4279	18013	3900	5775	5777	5793	11498
8000	4160	4162	4165	4539	18533	4160	6160	6162	6186	11628
8500	4420	4422	4425	4799	19053	4420	6545	6547	6595	11780
9000	4680	4682	4685	5059	19573	4680	6930	6932	6980	12295
9500	4940	4942	4945	5319	20093	4940	7315	7317	7365	12862
10000	5200	5202	5205	5643.5	20613	5200	7700	7702	7760	14040
10500	5460	5462	5465	6065	21133	5460	8085	8087	8343	14525
11000	5720	5722	5725	6372	21653	5720	8470	8472	8737	17315
11500	5980	5982	5986	6632	22173	5980	8855	8857	9122	20293
12000	6240	6242	6246	6892	32173	6240	9249	9242	9533	20553
12500	6500	6502	6506	7152	32693	6500	9625	9627	10108	21970
13000	6760	6762	6766	7428	33173	6760	10010	10014	10710	22480
13500	7020	7022	7026	7688	33433	7020	10395	10399	11330	22990
14000	7280	7282	7286	7948	33693	7280	10780	10784	11947	23500
14500	7540	7542	7546	8208	33953	7540	11165	11169	12551	24010
15000	7800	7802	7808	8468	34213	7800	11550	11554	13205	24520
15500	8060	8062	8068	8872	34473	8060	11935	11939	13817	25030
16000	8320	8322	8328	9318	34733	8320	12320	12324	14432	25540
16500	8580	8582	8588	9787	34993	8580	12705	12709	15121	26050
17000	8840	8842	8848	10465	35253	8840	13090	13094	15730	26560
17500	9100	9102	9108	10952	35733	9100	13475	13479	16254	27070
18000	9360	9362	9368	11451	36253	9360	13860	13864	16845	27580
18500	9620	9622	9628	11949	36773	9620	14245	14249	17440	28090
19000	9880	9882	9888	12447	58193	9880	14630	14634	18033	28600
19500	10140	10142	10148	12947	58453	10140	15015	15019	18636	30747
20000	10400	10402	10408	13457	60233	10400	15401	15404	19352	31262

7 Related Work

In this section, we introduce several similar frameworks to our system.

7.1 Condor

Condor [37] has a similar concept and architecture as the job submission mechanism of the queue in the Cyberaide Shell [4]. Condor provides a *pool* as the computing resources, including its batch scheduling system. In Condor Pool [37], there are three components that enable job execution with their cooperative operations: agents, matchmakers, and resources. When an agent wants to run a job on a resource, the matchmaker maps the agent to one of the resources based on the policy and job requirement information from the agent. The agent then can run a job on the resource.

Workflow management can be achieved with the Directed Acyclic Graph Manager (DAGMan) [37] service. Based on the script written by a user, DAGMan can set dependencies between jobs and execute them.

7.2 Swift

Swift [38] [39] integrates Karajan [40] and Falkon [41] for the execution engine and high throughput of fine-grained tasks. The Swift system presents a dynamic workflow in which the executions are determined at run time, which includes selecting resources, grouping jobs for optimization, and file staging. It also provides fault tolerance based on retrying and error handling. When the system experiences some faulty processing, it reschedules the tasks on the different resources or on the different sites after a number of failures on the current site. In addition, Swift can resume failed tasks in a workflow by checking a log that keeps information about the progress.

7.3 Fault Tolerance and Recovery in LEAD

Linked Environments for Atmospheric Discovery (LEAD) [42] is a research project that investigates weather changes such as floods or tornadoes threatening humanity. It includes the processing

of meteorological data with the need of large computational power and data management. It thus requires on-demand solutions with strict deadlines with dynamic and adaptive workflows in the distributed environments.

The Fault-Tolerance and Recovery service described in [24] is integrated into the LEAD [42] system. Two strategies, over-provisioning and migration, are selectively used for each workflow in the model. It decides which fault-tolerant strategy and resource to use based on the estimations of application performance, batch queue wait times, network latency and bandwidth, deadline and success probability, resource reliability, and service availability. Over-provisioning is used to run the multiple replicas in parallel so the execution can succeed unless all the replicas fail. Migration is used to run an application on another resource when a failure occurs. The application is executed again from the last checkpoint.

7.4 Pegasus

Pegasus [43] introduces a mechanism that maps an abstract workflow onto a distributed system. It supports dynamic mapping so that the workflow can be adjusted at run time to optimize the workflow in the changing environments. In addtion, the system can reduce queue wait time by clustering a number of jobs as one unit in case the jobs are scheduled to run on the same resource, so that the multiple jobs clustered can "share" one queue wait time rather than having separate wait time for each job.

In this system, fault tolerance is also integrated in such a way that if a failure occurs, the workflow is regenerated and mapped to the available resources.

7.5 Coasters

Coasters [44] show a mechanism that can avoid multiple queue wait times for fine-grained jobs. A service unit "occupies" a resource in time and space, so that a number of jobs can be directly submitted to the resource without staying in the local queue.

Coasters [44] provide three different services of *bootstrap*, *messaging*, and *coaster*. On the

client side, the *bootstrap* service initiates the Coaster system. After that, the *messaging* service is started on the client side for communication with the *coaster* service on the remote side at the later stage. The *coaster* service is finally activated on the remote site. Through direct communication between the *messaging* service unit and the *coaster* service unit, a number of jobs can be sent to the resource and executed in a sequential manner.

7.6 Other Similar Systems

In the system model in [23], the resource broker detects a node failure by checking the "signal" from the executing jobs. If there is no "signal" from the jobs, then it is assumed that the node has crashed. The fault occurrence information is then stored in the Grid Information Server (GIS) in order for the system to use the information for the future use of the resource, which includes deciding its fault tolerance strategy to be used when running jobs.

One scheduling strategy in [33] introduces a checkpointing like strategy with rerunning a failed job. This scheduling is designed for fine-grained tasks which applies a divide-and-conquer algorithm. The system has an information table that stores the results of the finished "orphan" jobs of which "parent" job in the tree hierarchy is failed by the processor crash. When the "parent" job crashes, the system restarts the computation but does not include the tasks already done which are stored in the table before the crash.

7.7 Summary & Discussion

The systems introduced in the previous sections provide fault-tolerant mechanisms for scientific workflows and clustering to reduce queue wait times. However, those systems handle failures with local checkpointing or rerunning jobs without considering site outages. Although the Swift system [38] schedules jobs at a multi-site level, it also does not incorporate site outages within the system. Additionally, the grouping strategies used in those systems to reduce the wait time in the local queuing systems have a different approach from ours. Firstly, we divide the computation and distribute them as multiple jobs on multiple sites to avoid a job failure from site outages. It is the

opposite way to grouping jobs to run them on the same resource. Secondly, our reducing queue wait time is based on the dynamic job dependency strategy. As jobs are distributed on multiple sites, clustering jobs is essentially not possible. Therefore, we devise dynamic job dependency which decides the dependency at run time, so that the queue wait times on the jobs can be "switched" depending on the job start time.

8 Limitation and Improvement Possibilities

Although the system is designed for the dynamic workflow depending on the queue wait time with fault tolerance, the current system has some limitations.

As described in Section 6.7, the dynamic fault-tolerant workflow has performance degradation due to the fact that each machine used for the generation distribution has the different compute performance.

To deal with this problem, an algorithm can be developed to select the best performance machine at run time. The computation can "migrate" from a worse compute performance machine to the better compute performance machine whenever it becomes available. To enable this strategy, a mechanism that a generation range can be assigned to the running jobs at run time is required. For example, if a job on a worse compute performance machine starts earlier than a better compute performance machine, the worse compute performance machine computes the generations until the job on a better compute performance machine starts. After the job on the better compute performance machine starts, the workflow engine can inform the current running job about it, so that the job can store the "mid-result" and send it to the new starting job for faster computation.

In a theoretical viewpoint, the generation distribution (specifying the generation range) can be seen as application-level checkpointing. This aspect can be applied to assigning the generations to different jobs on the best compute performance machine at run time whenever possible while not losing the idea of "distributing" the generations.

In addition, we have experienced a case in which the information on the TeraGrid Information Services [34] is not very accurate. The published information on the site outage time was different from the actual time of the outage. We expect that the information service will become more reliable in the future. Also, although our experience reveals that the gatekeeper node mostly does not go down during the site outage, it is a problem that the failure of the gatekeeper can cause the corrupted operation on the corresponding site which is hosting the gatekeeper since this queuing system largely relies on the GRAM [26] service through the gatekeeper node. This kind of problem cannot be solved within a single service of a system. Cooperated efforts on the multiple

components are required to build a reliable Grid system.

9 Conclusion

For the dynamic and fault-tolerant deployment of the threat management application, we have developed a queuing system that manages fault tolerance with the dynamic job submission strategy. This fault-tolerant queue communicates with Grid middlewares as the queue is built in the Cyberaide Shell [4] for its job submission, and it also integrates the TeraGrid Information Services [34] for the failure management. By distributing the generation processing on multiple sites, the system can avoid failures or delay of the processing from the site unavailability, so that it can deliver the services in real-time. In addition, our workflow design may reduce the queue wait time which causes an undesirable delay to the delivery of the results.

References

[1] G. von Laszewski, K. Mahinthakumar, R. Ranjithan, D. Brill, J. Uber, K. Harrison, S. Sreepathi, and E. Zechman, "An Adaptive Cyberinfrastructure for Threat Management in Urban Water Distribution Systems," in *Proceedings of the International Conference on Computational Science, ICCS 2006*, vol. 3993, 2006, pp. 401–408.

[2] S. Sreepathi, "Cyberinfrastructure for Contamination Source Characterization in Water Distribution Systems," Master's thesis, North Carolina State University, December 2006.

[3] C. e. a. Catlett, "TeraGrid: Analysis of Organization, System Architecture, and Middleware Enabling New Types of Applications," in *HPC and Grids in Action*, ser. Advances in Parallel Computing, L. Grandinetti, Ed. Amsterdam: IOS Press, 2007.

[4] G. von Laszewski, A. Younge, X. He, and F. Wang, "Cyberaide shell: Interactive task management for grids and cyberinfrastructure," http://cyberaide.googlecode.com/svn/trunk/papers/08-gridshell/vonLaszewski-08-gridshell.pdf, laszewski@gmail.com.

[5] G. von Laszewski, "The Grid-Idea and Its Evolution," *Journal of Information Technology*, vol. 47, no. 6, pp. 319–329, Jun. 2005.

[6] I. Foster, C. Kesselman, and S. Tuecke, "The Anatomy of the Grid: Enabling Scalable Virtual Organizations," *Intl. J. Supercomputer Applications*, vol. 15, no. 3, 2001.

[7] I. Foster and C. Kesselman, Eds., *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, Jul. 1998.

[8] D. Anderson, "BOINC: A System for Public-Resource Computing and Storage," in *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, 2004, pp. 4–10.

[9] "Portable Batch System." [Online]. Available: http://www.openpbs.org/

[10] I. Foster, "What is the grid? a three point checklist," June 2002. [Online]. Available: http://www-fp.mcs.anl.gov/~foster/Articles/WhatIsTheGrid.pdf

[11] ——, *A Globus Primer: Describing Globus Toolkit 4*, Globus, August 2005. [Online]. Available: http://www.globus.org/toolkit/docs/4.0/key/GT4_Primer_0.6.pdf

[12] "Web services architecture," [Online], http://www.w3.org/TR/ws-arch/.

[13] K. Kantardjieff, "The nsf cyberinfrastructure vision for 21st century discovery," [Online], 2007, http://www.calstate.edu/AcadSen/Newsletter/2007/tasn3_1007.pdf.

[14] V. Welch, I. Foster, C. Kesselman, O. Mulmo, L. Pearlman, S. Tuecke, J. Gawor, S. Meder, and F. Siebenlist, "X.509 Proxy Certificates for Dynamic Delegation," in *In Proceedings of the 3rd Annual PKI R&D Workshop*, 2004.

[15] J. Novotny, S. Tuecke, and V. Welch, "An Online Credential Repository for the Grid : MyProxy," in *Proceedings of HPDC 2001*, 2001.

[16] The Globus Security Team, "Globus Toolkit Version 4 Grid Security Infrastructure: A Standards Perspective," 2005. [Online]. Available: http://www.globus.org/toolkit/docs/4.0/security/GT4-GSI-Overview.pdf

[17] S. Zhao, A. Aggarwal, and R. D. Kent, "PKI-Based Authentication Mechanisms in Grid Systems," in *Networking, Architecture, and Storage, 2007. NAS 2007. International Conference on*, 2007, pp. 83–90.

[18] "Security Assertion Markup Language (SAML)," Web Page. [Online]. Available: http://www.oasis-open.org/committees/security/

[19] "Grid Security Infrastructure," http://www.globus.org/security/, Oct. 2003. [Online]. Available: WebPage

[20] E. M. Zechman and S. R. Ranjithan, "Niched Co-Evolution Strategies to Address Nonuniqueness in Engineering Design," in *Genetic and Evolutionary Computation Conference (GECCO)*. ACM, 2006.

[21] L. Rossman, "EPANET 2 users manual," US Environmental Protection Agency, Cincinnati, Ohio, Tech. Rep., 2000.

[22] W. Gropp, E. Lusk, and R. Thakur, *Using MPI-2 Advanced Features of the Message Passing Interface*, ser. Scientific and Engineering Computation. MIT Press Cambridge, MA, USA, November 1999.

[23] B. Nazir and T. Khan, "Fault Tolerant Job Scheduling in Computational Grid," in *Emerging Technologies, 2006. ICET '06. International Conference on*, November 2006, pp. 708–713.

[24] G. Kandaswamy, A. Mandal, and D. A. Reed, "Fault Tolerance and Recovery of Scientific Workflows on Computational Grids," in *Cluster Computing and the Grid, 2008. CCGRID '08. 8th IEEE International Symposium on*, May 2008, pp. 777–782.

[25] "TeraGrid User & System News," Web Page. [Online]. Available: http://news.teragrid.org/

[26] "GT 4.0 WS GRAM." [Online]. Available: http://globus.org/toolkit/docs/4.0/execution/wsgram/

[27] H. Li, "Workload Characterization, Modeling, and Prediction in Grid Computing," Ph.D. dissertation, LIACS, Computer Systems Group, Faculty of Science, Leiden University, 2008.

[28] "Lcg," [Online], http://lcg.web.cern.ch/LCG/.

[29] D. C. Nurmi, J. Brevik, and R. Wolski, "Qbets: queue bounds estimation from time series," in *SIGMETRICS '07: Proceedings of the 2007 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*. New York, NY, USA: ACM, 2007, pp. 379–380.

[30] "GSISSH." [Online]. Available: GSISSH.http://www.teragrid.org/userinfo/access/ssh.php#gsissh

[31] "Teragrid portal," [Online], http://www.teragrid.org/userinfo/portal.php.

[32] L. Ramakrishnan and D. A. Reed, "Performability Modeling for Scheduling and Fault Tolerance Strategies for Scientific Workflows," in *Proceedings of the 17th international symposium on High performance distributed computing*. Boston, MA, USA: ACM, June 2008, pp. 23–34.

[33] G. Wrzesiska, R. V. V. Nieuwpoort, J. Maassen, T. Kielmann, and H. E. Bal, "Fault-tolerant Scheduling of Fine-grained Tasks in Grid Environments," in *International Journal of High Performance Applications*, vol. 20, no. 1, February 2006, pp. 103–114.

[34] "TeraGrid Information Services," Web Page. [Online]. Available: http://info.teragrid.org/

[35] "The Monitoring and Discovery Service." [Online]. Available: http://www.globus.org/mds

[36] G. von Laszewski, "Java CoG Kit Workflow Concepts for Scientific Experiments," 2005.

[37] D. Thain, T. Tannenbaum, and M. Livny, "Distributed Computing in Practice: The Condor Experience," *Concurrency and Computation: Practice and Experience*, vol. 17, no. 2-4, pp. 323–356, 2005.

[38] Y. Zhao, M. Hategan, B. Clifford, I. Foster, G. von Laszewski, I. Raicu, T. Stef-Praun, and M. Wilde, "Swift: Fast, Reliable, Loosely Coupled Parallel Computation," in *IEEE Workshop on Scientific Workflows*, 2007.

[39] Y. Zhao, I. Raicu, I. Foster, M. Hategan, V. Nefedova, and M. Wilde, "Realizing Fast, Scalable and Reliable Scientific Computations in Grid Environments," *Grid Computing Research Progress, Nova Publisher*, 2008.

[40] G. von Laszewski, M. Hategan, and D. Kodeboyina, "Java CoG Kit Workow," *Workflows for e-Science, Springer*, pp. 340–356, 2007.

[41] I. Raicu, Y. Zhao, C. Dumitrescu, I. Foster, and M. Wilde, "Falkon: a Fast and Light-weight tasK executiON framework," in *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, 2007.

[42] K. K. Droegemeier, V. Chandrasekar, R. Clark, D. Gannon, S. Graves, E. Joseph, M. Ramamurthy, R. Wilhelmson, K. Brewster, B. Domenico, T. Leyton, V. Morris, D. Murray, B. Plale, R. Ramachandran, D. Reed, J. Rushing, D. Weber, A. Wilson, M. Xue, and S. Yalda, "Linked environments for atmospheric discovery (LEAD): A cyberinfrastructure for mesoscale meteorology research and education," in *20th Conf. on Interactive Information Processing Systems for Meteorology, Oceanography, and Hydrology*. Citeseer, 2004.

[43] E. Deelman, "Pegasus: A framework for mapping complex scientific workflows onto distributed systems," *Scientific Programming*, vol. 13, no. 3, pp. 219–237, 2005.

[44] "Coasters," Web Page. [Online]. Available: http://wiki.cogkit.org/wiki/Coasters

[45] "CoG Shell," Web Page. [Online]. Available: http://cogkit.svn.sourceforge.net/viewvc/cogkit/trunk/

A Appendix

A.1 Cyberaide Shell Installation Instruction

The initial version of the Cyberaide Shell [4] was conceptually introduced and practically demonstrated by Dr. Gregor von Laszewski at Argonne National Laboratory. The work is now continued at Indiana University. At the time the project was known under the name CoG Shell [45]. CoG Shell code was originally maintained at

http://cogkit.svn.sourceforge.net/viewvc/cogkit/trunk/

To download and install the Cyberaide Shell program, you need Maven installed on your machine. You can checkout the Cyberaide Shell program with the following command from the sourceforge svn:

```
> svn co https://cyberaide.svn.sourceforge.net/svnroot/\
cyberaide/code/trunk cyberaide
```

To compile and build the Cyberaide client program, run the following command in the cyberaide directory:

```
> make clean; make copy; make build
```

To run the Cyberaide client program, run the following command in the cyberaide directory:

```
> make cli
```

You can find the related document on the following webpage:

http://cyberaide.org/tmp/cyberaide-developers-manual

A.2 Other Source Code & Data

Other source code for the simulation and outage data can be found in the following url:

http://cyberaide.googlecode.com/svn/trunk/papers/thesis-moon/outage/

Please note that the source code for the Water Threat Management [1] [2] application is not publicly available.

A.3 Cyberaide Shell Queue Command Usage

```
SYNOPSIS

  queue -list
  queue -submit -command program [argument ...] [-mpi number] [-directory
    directory] [-inputfile filename] [-outputfile filename] [-stdout
 filename] [-stderr filename]
  queue -stat -job-id jobid
  queue -suspend -job-id jobid
  queue -resume -job-id jobid
  queue -cancel -job-id jobid
  queue -policy [-task [-default|-replicate|-random]] [-resource [-default|
    -random]]

DESCRIPTION

  The queue command creates and submits a task to the task pool with policies.
  A task and a resource is mapped based on the task and resource policies, and
  the task is scheduled for execution on the TeraGrid resources.

OPTIONS

  -list
   Lists currently available TeraGrid resources hosting GRAM4 service.
```

```
-submit
 Submit a job to the task pool.

-inputfile filename
 Specify an input file for the program.

-outputfile filename
 Specify an output file for the program.

-errorfile filename
 Specify an error file for the program.

-command program [argument ...]
 Specify the executable and arguments if the arguments are needed.

-directory directory
 Specify a directory for a job.

-stat
 Check job status.

-suspend
 Suspend a job.

-resume
 Resume a suspended job.

-cancel
 Cancel a job.

-job-id jobid
 Specify a job id generated in Cyberaide queue.
```

-policy

Set a policy for the task pool / resource pool for job scheduling.

-default

Default policy for the task pool (FIFO) and resource pool (Round-Robin).

-replicate

Fault-tolerant policy for the task pool. When this policy is set, every task submitted to the TeraGrid resource is replicated.

-random

Random policy for the task pool and resource pool. A task and resource are selected randomly.