Grid Workflow:
A Flexible Failure Handling Framework for the Grid

Soonwook Hwang and Carl Kesselman
Information Sciences Institute
University of Southern California
Marina del Rey, CA 90292
{hwangsw,carl}@isi.edu

Abstract

The generic, heterogeneous, and dynamic nature of the Grid requires a new form of failure recovery mechanism to address its unique requirements such as support for diverse failure handling strategies, separation of failure handling strategies from application codes, and user-defined exception handling. We here propose a Grid Workflow System (Grid-WFS), a flexible failure handling framework for the Grid, which addresses these Grid-unique failure recovery requirements. Central to the framework is flexibility in handing failures. We describe how to achieve the flexibility by the use of workflow structure as a high-level recovery policy specification. We show how this use of high-level workflow structure allows users to achieve failure recovery in a variety of ways depending on the requirements and constraints of their applications. We also demonstrate that this use of workflow structure enables users to not only rapidly prototype and investigate failure handling strategies, but also easily change them by simply modifying the encompassing workflow structure, while the application code remains intact. Finally, we present an experimental evaluation of our framework using a simulation, demonstrating the value of supporting multiple failure recovery techniques in Grid systems to achieve high performance in the presence of failures.

1 Introduction

The Grid environment refers to the Internet-connected computing environment in which computing and data resources are geographically dispersed in different administrative domains with different policies for security and resource uses; and the computing resources are highly heterogeneous, ranging from single PCs and workstations, cluster of workstations, to supercomputers. With Grid technologies it is possible to construct large-scale applications over the Grid environment [9, 4]. However, developing, deploying, and running applications on the Grid environment poses significant challenges due to the diverse failures and error conditions encountered during execution.

Failures or error conditions due to the inherently unreliable[1] nature of the Grid environment include hardware failures (e.g., host crash, network partition, etc), software errors (e.g., memory leak, numerical exception, etc) and other sources of failures (e.g., machine rebooted by the owner, network congestion, excessive CPU load, etc). Beside the need to deal with this unreliable nature of the Grid environment, Grid applications should be able to handle failures which are sensitive to the task context, what we call *task-specific failures*.

Grid applications are in general distributed, heterogeneous multi-task applications where the component tasks integrated into the Grid applications could be implemented by arbitrary applications. Each task has its own failure semantics; that is, failure definition and failure handling strategies are specific to the task. For example, a linear solver task should reach convergence within 30 minutes; otherwise, it would be considered to be a performance failure because something unexpected happens, such as excessive CPU load, or the priority of the linear solver lowered by the owner of the host for his own jobs. As another example, a simulation task requires a certain amount of disk space to save temporary results. If there is not enough disk space remaining, the simulation task will fail due to the lack of disk space.

These and other types of task-specific failures as well as failures in the Grid environment should be able to be detected and handled in a variety of ways depending on the execution semantics of both the task and the overall Grid

[1] The Grid environment is unreliable because it is geographically dispersed; it involves multiple autonomous administrative domains; it is composed of a large number of components (e.g., instruments, display, computational and informational resource, people).

Proceedings of the 12th IEEE International Symposium on High Performance Distributed Computing (HPDC'03)
1082-8907/03 $17.00 © 2003 IEEE

application:

- In case of the linear solver, if not completed within 30 minutes, terminate the task, allocate a new resource, and restart;
- In case of the simulation task, if not enough disk space remaining, terminate the task in advance, and either restart it on a machine with significantly more disk space, or retry it on the same machine, but with a different algorithm which requires less disk space;
- In case of a long running task, checkpoint periodically and restart from the last good state;
- In case of a task running on unreliable execution environments, have multiple replicas of the task run on different machines, so that as long as not all replicated tasks fail, the task will succeed to execute;
- In other case, undo the effect of the failed task and retry, or ignore the failure and continue, etc

Existing distributed systems, parallel systems, or even so-called Grid systems designed to address fault tolerance issues fail to address the Grid-specific failure recovery issues such as task-specific failure handling and multiple failure handling schemes. As far as we know, existing systems focus on integrating only one failure-type independent fault tolerance technique as for their failure handling policy. For example, in the traditional distributed systems such as large-scale transactional distributed systems, even though dispersed geographically on wide area networks, the component tasks[2] are not heterogeneous. Instead, the component tasks are not only homogeneous but also simple (i.e., mainly read and write date item), and thus need not distinguish the type of failures returned. As a result, the transaction-based recovery (i.e., logging and rollback), a failure-type independent fault tolerance technique, appears to be sufficient as only one failure handling policy for the transactional distributed system. However, for the Grid applications which consist of arbitrary component tasks, each with its own failure semantics, the failure-type independent fault tolerance technique does not work well because it can support neither task-specific failure definition, detection, and handling nor diverse failure handling strategies.

In this paper, we present a flexible failure handling framework which addresses the unique requirements for fault tolerance in the Grid such as support for diverse failure handling strategies, separation of failure handling strategies from application codes, and task-specific failure handling. The framework allows users to achieve failure recovery in a variety of ways based on the requirements or constraints of their applications following failure detection. The core of our framework is the use of *workflow* [16, 13, 20, 17] as a recovery strategy specification to specify diverse failure recovery procedures at a high level rather than hardcoding them inside application codes.

A *workflow* enables the structuring of applications in a directed acyclic graph form, what is called workflow structure, where each node represents the constituent task and edges represent inter-task dependencies of the applications. We incorporate two-level failure recovery into the workflow structure:

- *Task-level* techniques refer to recovery techniques that are to be applied in the task level to mask the effect of *task crash* failures. These techniques realize the so-called *masking fault tolerance* [11] techniques such as *retying*, *checkpointing*, and *replication*.
- *Workflow-level* techniques refer to recovery techniques that enable the specification of failure recovery procedures as part of application structure. Thses techniques realize the so-called *nonmasking fault tolerance* [11, 8] techniques such as *alternative task*; basically, these techniques allow alternative tasks to be launched to deal with not only *user-defined exceptions* but also the failures that *task-level* techniques fail to mask (e.g., due to not enough redundant resources) in the task level.

We show how users can specify diverse failures handling strategies (e.g., depending on the performance goal of their applications, the availability of Grid resources, task-specific execution semantics, etc) with *task-level* techniques, *workflow-level* techniques, and even the combination of both techniques. We also demonstrate the flexibility of our framework by describing how it allows users to rapidly prototype, investigate different fault tolerant schemes, and more importantly, to easily change them according to the changes in the underlying Grid structure and state. We have prototyped a Grid Workflow System (GridWFS) which implements all the features described above. We present an experimental evaluation of our framework using a simulation, demonstrating the value of supporting multiple failure recovery techniques in Grid systems to achieve high performance in the presence of failures.

2 Requirements for Failure Recovery Mechanism on the Grid

We have identified three requirements for Grid failure recovery mechanisms. Basically, these requirements are driven by the unique properties of the Grid such that Grid environments are generic, heterogeneous and dynamic, and Grid applications are heterogeneous.

[2] In traditional distributed systems, the component tasks are usually named as *individual services* comprising the distributed system.

2.1 Support for Diverse Failure Handling Strategies

We require that the Grid failure recovery mechanism support a wide range of failure handling strategies following the detection of failures. This requirement is driven by the heterogeneous nature of the Grid context, such as heterogeneous tasks (e.g., long running tasks, mission critical tasks, transactional tasks, etc) and heterogeneous execution environments (e.g., highly reliable execution environments such as the Condor resource, unreliable execution environments such as a single workstation donated by an anonymous volunteer for its idle computing cycle, etc).

This heterogeneity aspect introduces a need for a flexible failure handling mechanism that supports multiple fault tolerance techniques, allowing each task to select an appropriate fault tolerance technique among alternatives depending on e.g., the task characteristic, or an estimated reliability of the underlying execution environment. For example, suppose that a Grid computing resource on which a task is running has a long downtime, the task may prefer the "retrying on another available Grid resource" strategy to either the "retrying on the same resource" or "restarting with checkpointing on the same resource" strategy.

2.2 Separation of Failure Handling Policies from Application Codes

We require that the Grid failure recovery mechanism enable the separation of failure handling policies from application codes. This requirement is driven by the dynamic nature of the Grid. The state and structure of the underlying Grid is constantly changing. For example, software resources with a new novel algorithm are added. New hardware resources are added and old ones are retired.

In order for users to address the constantly changing nature of the Grid, a high-level specification for failure handling should be supported so that they can specify failure handling policies at a high level without having to hardwire them within the application algorithm. As a result, users should be able to quickly adapt failure handling policies to the newly changed Grid environments simply by modifying the high-level policy description. Coding manually those task-specific failure detection and failure handling procedures within the application is not a viable solution because it makes the design and development of Grid applications much more complicated; furthermore, this approach requires application programmers to start from scratch by embedding fault tolerance procedures inside the application code in an *ad hoc* manner each time they develop a new application.

2.3 User-defined Exception Handling

We require that the Grid failure recovery mechanism support *user-defined exception* handling. That is, users should be allowed to specify *user-defined exceptions* to handle *task-specific failures* base on the task context. In addition, users should be able to specify appropriate exception handling procedures to deal with the associated *user-defined exceptions* occurring during task execution.

For example, suppose that we have a computation that needs to be accomplished, for which there are two algorithms available. One algorithm is faster than the other, but requires a large amount of memory. The other claims less memory by using a local disk instead of memory, but is slower than the first. A task using the first algorithm may fail during execution due to "out of virtual memory space." In this case, users should be able to specify a user-defined exception called "out of memory" for the task using the first algorithm. In addition, users should be allowed to specify a failure handling policy like: *if a task using the first algorithm fails due to the "out of memory" exception, try an alternative task using the second algorithm rather than retrying the same task.*

3 Overview of our Approach

In this section we describe a failure recovery mechanism for the Grid designed to meet the requirements mentioned above. Figure 1 presents an overview of our approach to fault tolerance in the Grid, which comprises two phases (i.e., failure detection and recovery phase) represented by *a generic failure detection mechanism* and *a flexible failure handling framework*. The generic failure detection service designed to be used in Grid applications relies on heartbeat and event notifications to enable the detection of two failure classes (i.e., the *task crash* failure and *user-defined exceptions*) during task execution on Grid nodes. That is, on receipt of heartbeat and event notification massages being delivered from each Grid node, Grid applications can interpret these messages to determine the state (i.e., *"inactive", "active", "done", "failed", "exception"*) of their component tasks submitted to the Grid node. In addition, the service allows users to specify user-defined exceptions to handle task-specific failures. In [18], we have described more details on our generic failure detection service for the Grid, including failures of concern, its architecture, the task state transition, the format of notification messages, the task-specific event notification API which is to be called inside the component task codes to send event notifications (e.g., "Task End", "Exception") to Grid applications, etc.

The flexible failure handling framework uses workflow structure in which we have integrated *task-level* and *workflow-level* failure recovery techniques. The *task-level*

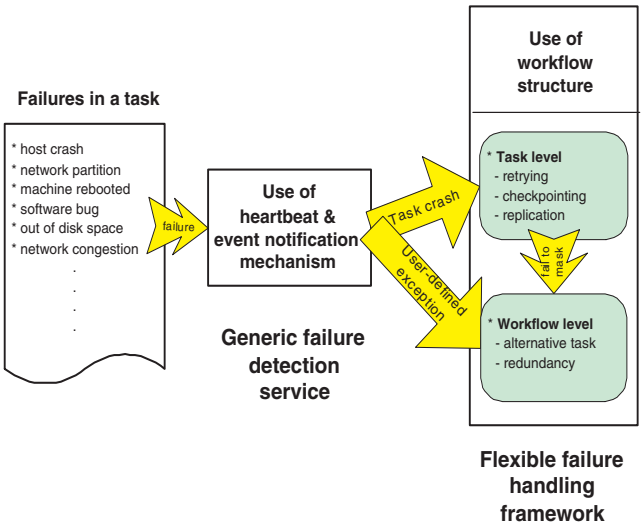

Figure 1. Overview of our approach

techniques are intended to mask *task crash* failures in the task level, i.e., without affecting the flow of the encompassing workflow structure. The *workflow-level* techniques, on the other hand, are intended to apply in the workflow level (i.e., by allowing changes to the flow of workflow execution) to handle the *user-defined exceptions* as well as the *task crash* failures that cannot be masked with the *task-level* techniques. With these two-level techniques, users can specify failure handling strategies at different levels of abstraction; the *task-level* techniques enable users to mask generic task failures (i.e., failure-type independent failures) without having to know about the task context, while the *workflow-level* techniques enable users to define an appropriate recovery procedure in the application structure to handle task-specific failures (i.e., failure-type sensitive failures) based on their knowledge about the task execution context.

Note that in the Figure 1, the arrow with the label of "Task crash" on it indicates that the *task crash* failure detected by the *generic failure detection mechanism* can usually be handled with the *Task level* techniques such as retrying, checkpointing, and replication. Similarly, the arrow with the label of "User-defined exception" denotes that the *user-defined exceptions* can be dealt with by the workflow-level techniques such as the concept of alternative task, or workflow-level redundancy (e.g., launching several tasks with each implemented by different algorithms, in hoping that one of the redundant tasks will finish successfully). Another arrow connecting the *Task level* to the *Workflow level* is seen in the figure with the label of "fail to mask" on it; this one tells that in case of the *Task level* recovery techniques having failed to mask failures (e.g., due to not enough redundancy to mask them at the task level), the *workflow level* recovery techniques can be applied to deal with the propagated failures.

When users designing and implementing fault tolerant applications, this failure handling framework gives a great deal of flexibility by supporting a wide assortment of failure handling strategies, ranging from well-known task-level fault tolerance techniques to various user-specified application-level failure management schemes; to this end, the framework provides users with the XML Workflow Process Definition Language called XML WPDL (Section 7). This use of high-level workflow structure factors out the design of fault tolerance schemes from the low-level details of application algorithm design. Consequently, different fault tolerance schemes and designs can be rapidly prototyped, investigated, and more importantly can be easily changed according to the changes in the underlying Grid structure as described in the following sections.

4 Task-level Failure Handling Techniques

In this section we describe failure handling techniques that can be applied in the task level so as to prevent *task crash* failures from being propagated to the workflow level.

4.1 Retrying

This might be the simplest failure recovery technique to use in hope that whatever cause of the failures will not be encountered in subsequent retries. Figure 2 shows an example of workflow specification fragment using the XML WPDL. It describes that if the task crash failure is detected (i.e., by receiving *Done* without *Task End* notification as described in [18]), this particular task named "summation" would be retried on the specified Grid resource (i.e., whose hostname is "bolas.isi.edu") up to 3 times with an interval of 10 seconds between tries.

4.2 Replication

The basic idea of this failure handling technique is to have replicas of a task run on different Grid resources, so that as long as not all replicated tasks crash (due to host crash, host partition away from the Grid client, etc), the task execution would succeed. Figure 3 depicts an example of a XML WPDL fragment which specifies this particular task to be replicated onto three different Grid resources. Note that users can easily choose to use this technique simply by specifying the *policy='replica'* in the Activity definition for the task.

When the task, *summation*, is performed, the underlying system simultaneously submits the task execution request to three specified Grid resources. Once recognizing that one of

```
<Activity name='summation' max_tries='3'
                          interval='10'>
  <Input>...</Input>
  <Output>...</Output>
  <Implement>sum</Implement>
</Activity>
 ....
<Program name='sum'>
 <Option hostname='bolas.isi.edu'
   service='jobmanager'
   executableDir='/XML/EXAMPLE/'
   executable='sum'/>
</Program>
```

Figure 2. XML WPDL example of retrying. Users can specify the maximum number of retries and the interval between retries by using the *max_tries* and the *interval* attributes in the task specification. Note that this code represents *retrying on the same resource*. Users can also specify *retrying on different resources* by simply defining multiple Grid resources as seen in Figure 3.

```
<Activity name='summation'
                   policy='replica'>
   ...
  <Implement>sum</Implement>
</Activity>
   ...
<Program name='sum'>
 <Option hostname='bolas.isi.edu' .. />
 <Option hostname='vanuatu.isi.edu' .. />
 <Option hostname='jupiter.isi.edu' .. />
</Program>
```

Figure 3. XML WPDL example of replication. Users can specify a particular task to be replicated on multiple Grid resources by defining the *policy='replica'* in the task definition and multiple resources within the corresponding <Program> definition.

the submitted tasks has finished successfully by receiving both the *Done* and the *Task End* notification messages as described in [18], the system considers the task execution as having succeeded.

4.3 Checkpointing

Checkpointing has been studied a great deal in distributed, parallel systems as an efficient fault tolerance technique especially for long-running applications. As a result, to date many checkpoint libraries and program development libraries which support checkpointing are available. With these checkpointing facilities, checkpoint-enabled applications can be developed simply by linking to them. Dome [3], Fail-safe PVM [21], and CoCheck [25] are examples of such program development libraries that enable coordinated checkpointing on parallel computing platforms (e.g., network of workstations), while Libckpt [22] and the Condor checkpoint library [1] are standalone portable checkpoint libraries for uniprocessor platforms.

Our framework is designed to support checkpoint-enabled tasks. That is, when a task fails, it is allowed to be restarted from the recently checkpointed state rather than from the beginning. Users do not have to specify anything about the checkpointing (for example, specifying something like *policy='checkpoint'* as in the case of the replication technique) in the workflow structure. Instead, all they have to do is to call one of task-specific event notification API functions (i.e., *globus_FDS_task_checkpoint()*) [17] within the checkpoint-enabled task code so as to notify the framework that this task is a checkpoint-enabled task. Upon receipt of the checkpoint notification from a task, the framework marks the task as checkpoint-enabled, and saves the checkpoint flag being delivered piggybacked on the notification message. Hence, when the the *task crash* failure is detected and retrying is specified, the framework retries the task from the checkpointed state by sending back the checkpoint flag. Currently, we have successfully tested this checkpointing feature of our framework with the Libckpt standalone checkpoint library.

5 Workflow-level Failure Handling Techniques

In this section we describe failure handling techniques that could be applied in the workflow level. The manipulation of workflow structure (e.g., modifying execution flows to deal with erroneous conditions) is the basis of these techniques.

5.1 Alternative task

A key idea behind this failure handling technique is that when a task has failed, an alternative task is to be performed to continue the execution, as opposed to the *retrying* technique where the same task is to be repeated over and over again which might never succeed. This technique might be desirable to apply in some cases where there are two different task implementations available for a certain computation; each implementation, however, has different execution characteristics. For example, the first implementation runs fast, but is unreliable, whereas the second runs slow, but is reliable. In this case, users can specify the second implementation to act as an alternative task to the first one.

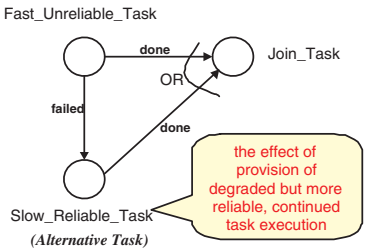

Figure 4. Task crash failure handling using an alternative task

Figure 4 illustrates an example of failure handling using the *alternative task* technique. In the figure, the *Slow_Reliable_Task* is specified as an alternative task to the *Fast_Unreliable_Task*, such that the *Slow_Reliable_Task* would be activated to recover from the *task crash* failure that might happen to the *Fast_Unreliable_Task* during its execution.

Note that this technique is also useful in cases where users wish to semantically undo the effect of a failed task. For example, for a task which transfers a huge amount of data, users may want to define an alternative task such that the alternative task is activated to clean up the partially transferred data if the original task has failed during execution.

5.2 Workflow-level redundancy

As opposed to the task-level replication technique where same tasks are replicated, having multiple different tasks run in parallel for a certain computation is the basic idea of this technique. Thus, as long as at least one task has finished successfully, then the computation would succeed. This technique might be useful in cases where there are many task implementations with different execution behavior available for the computation. For example, the first implementation is fast but unreliable, and the second slow but reliable. There might be other implementations available which have different execution behaviors than the first and the second. In this case, users may want to have all these different implementations simultaneously executed so as to e.g., achieve fault tolerance and/or performance goals at the cost of extra CPU consumption. This is simply achieved by specifying (1) a *split* task with multiple outgoing control flows to each of the different implementations whose transition condition evaluates always to true, (2) a *join* task with multiple incoming control flows from each of the implementations, and (3) an *OR* relationship between the incoming control flows (see Figure 5).

Figure 5 depicts an example of using the workflow-level redundancy technique to achieve a certain level of fault tolerance by having the two different task implementations (i.e., *Fast_Unreliable_Task* and *Slow_Reliable_Task*) run in parallel. Note that in the figure, the *OR* relationship between the two incoming control flows (i.e., one from the *Fast_Unreliable_Task* and the other from the *Slow_Reliable_Task*) indicates that the *Dummy_Join_Task* would be activated if either execution of the two tasks succeed.

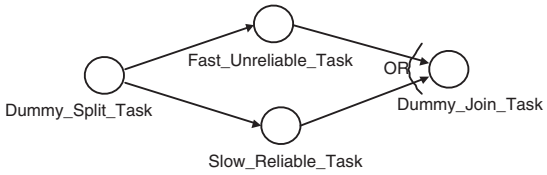

Figure 5. Workflow-level redundancy

5.3 User-defined Exception Handling

This technique allows users to give a special treatment to a specific failure of a particular task. This could be achieved by using the notion of the *alternative task* technique; that is, by specifying an workflow process such that the alternative task is to be launched to deal with the specific failure if the particular task fails due to the specific failure.

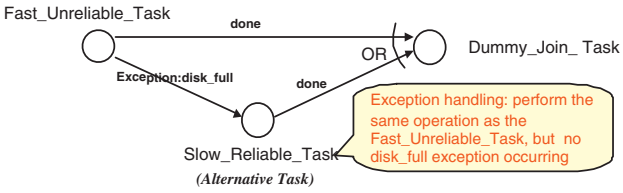

Figure 6. Exception handling using an alternative task

Figure 6[3] shows an example of user-defined exception handling using an alternative task; the *Slow_Reliable_Task* is specified to be activated to handle a task-specific failure (i.e., a user-defined exception called *disk_full*) that might arise during the execution of *Fast_Unreliable_Task*.

6 Flexibility of our Framework

Our framework is designed to be flexible enough to:

[3]Note that due to page limitation, we omitted WPDL workflow specification codes for Figure 4, 5, and 6. See [17] for details on the codes.

- *combine between task-level failure recovery techniques.* For example, in the code (Figure 3), users can specify each replica to be retried when it fails by just adding maximum number of retries into the activity definition.

- *combine between task-level and workflow-level failure recovery techniques.* For example, in Figure 4 and Figure 5, users can make the *Fast_Unreliable_Task* more tolerant to *task crash* failures by applying task-level failure recovery techniques such as *retrying*, *checkpointing*, and *replication*.

- *change failure handling strategies easily and incrementally by changing its workflow structure as the underlying Grid structure changes.* For example, as seen in Figure 4, Figure 5, Figure 6, with the two tasks (i.e., *Fast_Unreliable_Task*, *Slow_Reliable_Task*), users can structure different failure handling strategies, and can incrementally change them simply by changing the workflow structure. Thus, there is no need to recompile, relink, and test the application source codes as the failure handling strategies change.

7 Implementation

We have prototyped a Grid Workflow System (Grid-WFS) that implements the framework described above, The overall system structure of Grid-WFS is illustrated in the Figure 7, which consists of three major components:

- A Workflow Process Definition Language using XML (XML WPDL) that allows users to define workflow process specification in a Directed Acyclic Graph (DAG) form.

- A Workflow Engine that controls workflow execution by navigating the workflow specifications, submitting tasks to specified Grid nodes, and monitoring the status of submitted tasks.

- *Workflow runtime services* that provide directory services necessary for the workflow engine to perform resource brokering during the workflow execution, including software, data, and resource category services.

All features described in this paper can be specified using the XML WPDL. Furthermore, additional features are supported by the XML WPDL, including conditional transition (e.g., *if-then-else*), loop structure (e.g., *do-while*). Thus, users can specify more sophisticated application structure and application-level failure handling strategies (e.g., value dependency [17]). Further details on the specification, syntax (i.e., XML Document Type Definition) and example usages of XML WPDL can be found in [17].

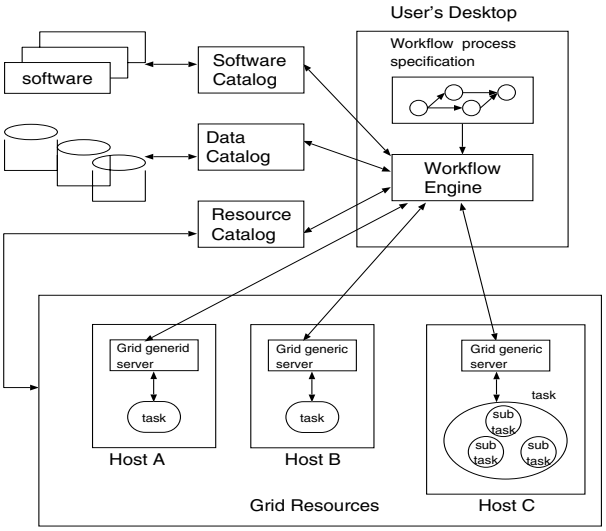

Figure 7. Grid-WFS architecture

The workflow engine is the core part of the prototype. It is implemented as a standalone application on top of the Globus Toolkit [2] v2.0. When the engine is started, it reads a workflow process specification from a file as specified in its input argument and creates an instance of the specification in a parse tree form. The engine, then, begins to navigate through the parse tree. The engine determines the tasks that are ready to execute by examining whether their dependencies has been resolved or not. Once having identified such tasks, the engine submits them to appropriate Grid resources via the Globus GRAM [6] protocol. The engine is designed to identify the appropriate Grid resources either as specified in the workflow specification or by consulting with the directory services.[4] The engine determines the final status (i.e., *done*, *failed*, or *exceptions*) of the submitted tasks using the generic failure detection mechanism described in [18], storing the final status in the parse tree. The engine then re-evaluates the parse tree, identifies the next tasks whose dependencies have been resolved, and then submit them. The navigation continues in this way through the workflow process instance until either it is completed successfully or is terminated unsuccessfully due to any unrecoverable erroneous situations. For fault tolerance of the workflow engine itself, we have implemented the checkpointing of the workflow engine. That is, every time a task termination state is recognized, the engine saves the current XML parse tree onto a persistent storage in a XML file form. So, when being restarted, the engine creates a parse tree from the saved XML file rather than from the original XML file and begins navigation from where it left off.

[4] We have not implemented the second option yet.

8 Evaluation

The Grid-WFS supports multiple failure recovery techniques as opposed to most other distributed (or even so-called Grid) systems in which only a single failure recovery technique is supported. In this section, we present an experimental evaluation of the Grid-WFS, demonstrating the value of supporting multiple failure recovery techniques in Grid environments to achieve high performance in the presence of failures.

8.1 Experimental Methods - Simulation

We measured using simulation the expected completion time of a task in spite of failures during its execution. Following are the parameters being used for our simulation: (mostly borrowed from [23, 3, 7])

- **Failure-free execution time (F)**. This is the execution time of a task in the absence of failure.
- **Failure rate** (λ). This is a random variable representing an arrival rate of failures governed by a Poisson distribution, as is commonly assumed in fault tolerance literature [23, 3, 7]. TTF (Time To Failure) is a random variable representing the time between adjacent arrivals of failures, governed by the well-known exponential distribution [19]. MTTF (Mean Time to Failure) is a mean TTF interval, mathematically defined by $1/\lambda$ [23].
- **Downtime (D)**. This is the average time following a failure of a task before it is up again, governed by the exponential distribution [23].
- **Average checkpoint overhead (C)**. This is the average amount of time required to create a checkpoint. We assume it a constant variable in our simulation.
- **Uninterupted task execution time between checkpoints (a)**. This is the time interval between two consecutive checkpoints in the failure-free runs [7, 3]. So, if K checkpoints are created during F, then a = F / K.
- **Recovery time (R)**. This is the time that it takes to restore the checkpointed state following the detection of a failure [23].
- **Number of replicas (N)**. This is the number of replicated tasks, with each running on different machines.

We note that the above parameters are not all parameters that need to be considered; checkpoint latency (L) [23] should also be taken into account for more precise simulation, but for simplicity, by assuming that a task is halted during checkpointing we do not consider this parameter in our simulation.

Based on the above parameters, we measured the expected execution time of tasks with four different types of failure recovery techniques:

- *Retrying.* The basic idea is that if a failure occurs, then a task must restart from the beginning. We simulate the completion time of a task based on the assumption and analysis in Duda [7] on program without checkpointing.
- *Checkpointing.* A task stores periodically its states. So, when the task crashes, it can restart from the most recently checkpointed state. The time needed to complete a task is simulated based on the assumption and analysis in Duda [7] on program with checkpointing.
- *Replication.* We calculate the completion time by running a task as many as N times and then choosing the smallest completion time among those obtained from the N simulation runs. Note that each run is assumed to employ the *retrying* as its recovery technique to continue to run until it has completed.
- *Replication w/ checkpointing.* The completion time is computed in the same way as used in the above *Replication* except that each run uses the *checkpointing* recovery technique to complete its computation.

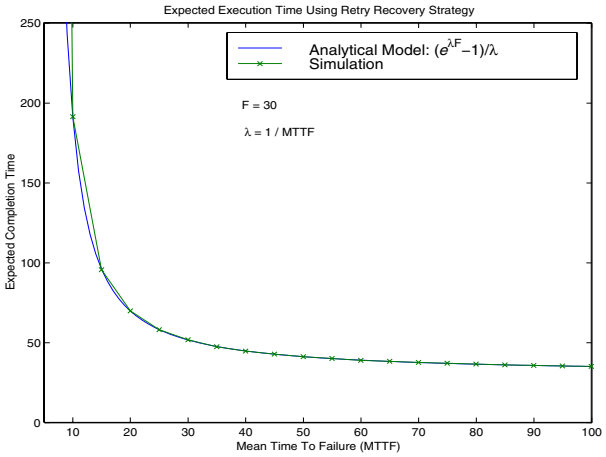

Figure 8. Comparison between analytical and simulation results for *retrying*

To validate the correctness of our simulation results, we compared them with analytical models from fault tolerance literature. Figure 8 and Figure 9 plot a graph of the expected

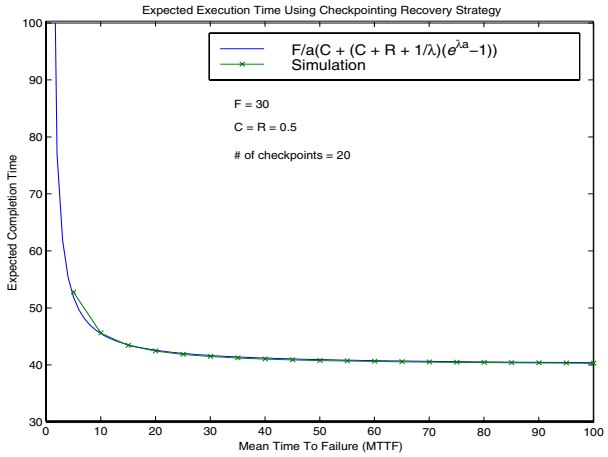

Figure 9. Comparison between analytical and simulation result for *checkpointing*

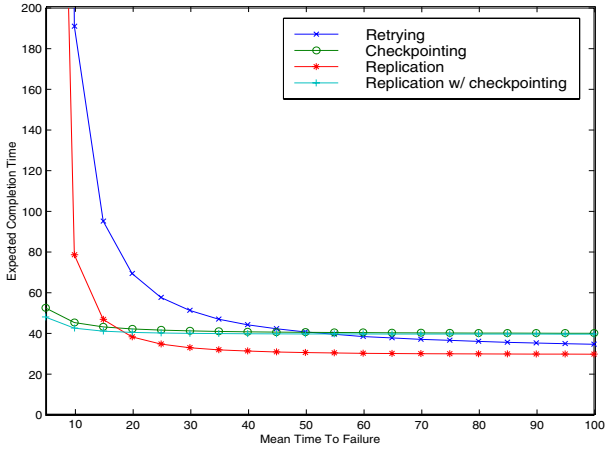

Figure 10. Comparison between fault tolerance techniques as MTTF increases. This graph shows the expected completion time as a function of MTTF.

completion time as a function of MTTF from analytical results[5] overlaid with our simulation results. As can be seen in both figures, the expected completion time from simulation results is the same as the analytical expected completion time, which proves that our simulation method is correct. Note that for this simulation results, we ran 10,000, 100,000 and 1,000,000 runs of simulations, and found out that 100,000 runs are enough for our simulation. In fact, the simulation results in both figures were obtained by running 100,000 runs of simulation.

8.2 Experimental Results

We first tested the influence of tast crash failure rate on the selection of fault tolerance techniques to achieve high performance in the presence of failures. We simulated the expected completion time of a task with different fault tolerance techniques for various failure rate. For this experiment, we fixed the parameter F to 30, K (i.e., number of checkpoints) to 20, D to 0, both C and R to 0.5, and N to 3. Figure 10 plots the simulation results. The figure shows that when task crash failure rate λ is high (i.e., the smaller value of MTTF), checkpointing and replication w/ checkpointing outperform the other two techniques. However, for smaller value of failure rate (i.e., greater value of MTTF), the use of checkpointing and replication w/ checkpointing mechanism as a recovery policy appears to be inappropriate due to the checkpoint overhead. If task execution environments are reasonably reliable, (i.e., MTTF is greater than 18, $\frac{MTTF}{F} > 0.6$), then replication performs better than any other techniques at the cost of extra CPU consumption.

[5]We obtain the analytical results from Plank [23] and Dome [3], respectively

We then tested the impact of downtime on the completion time of a task. The expected completion time was computed using the same parameters as the above experiment except for using various values of downtime: 0, F (30), 5F (150), and 10F (300). The experiment results are plotted in Figure 11, illustrating that in case of longer downtime, replication and replication w/ checkpointing perform better than the other two techniques. However, when failure rate is high, failure rate is a more dominant factor than long downtime. As can be seen in Figure 12 which zooms out the "downtime = 10F" graph, when failure rate is relatively high (i.e., MTTF is less than 12, $\frac{MTTF}{F} < 0.4$), checkpointing performs better than replication. Figure 12 also indicates that in low reliable (i.e., failure rate is high) and low available (i.e., downtime is long) execution environments, as we have expected, the strongest fault tolerance technique (i.e., replication w/ checkpointing) outperforms the other techniques.

We also tested the impact of whether or not to support user-defined exception handling mechanism on the performance of Grid applications. We computed the expected completion time of the exception handling DAG shown in Figure 6. Following are the assumptions that we have made for this experiment:

- The duration of the *Fast_Unreliable_Task* (FU), the *Slow_Reliable_Task* (SR), and the *Dummy_Join_Task* (DJ) are 30, 150, and 0, respectively.
- The FU checks five times during its execution (i.e., every 6) as to whether the disk full exception occurs or not. For simplicity, we model the FU as a Bernoulli

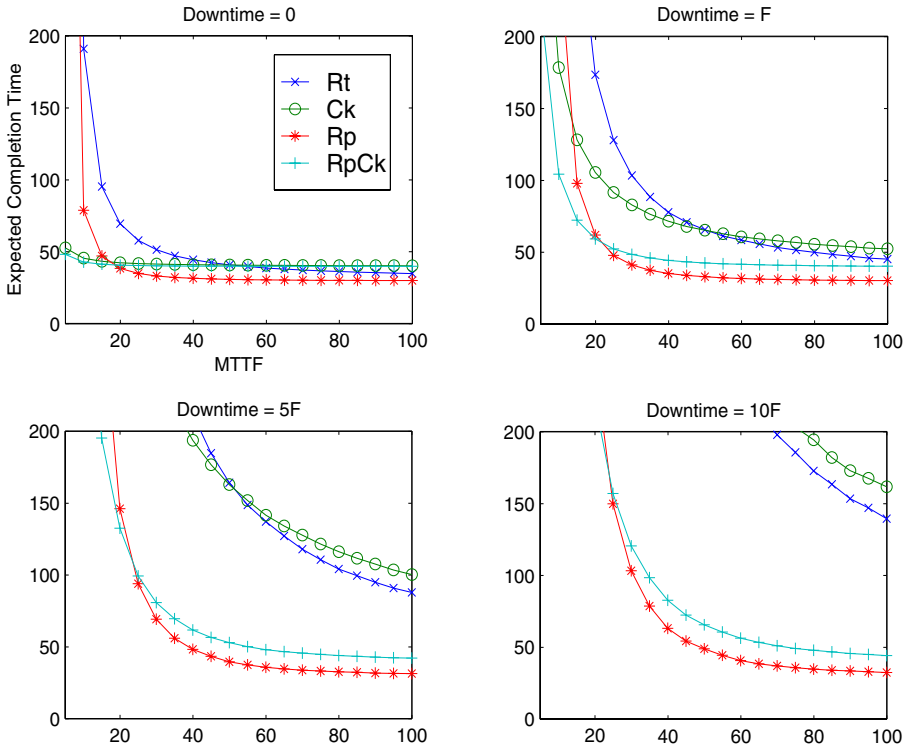

Figure 11. Comparison between fault tolerance techniques as Downtime increases. This graph shows the impact of downtime on the performance of different fault tolerance techniques. The legend Rt represents *Retrying*, Ck *Checkpointing*, Rp *Replication*, and RpCk *Replication w/ checkpointing*.

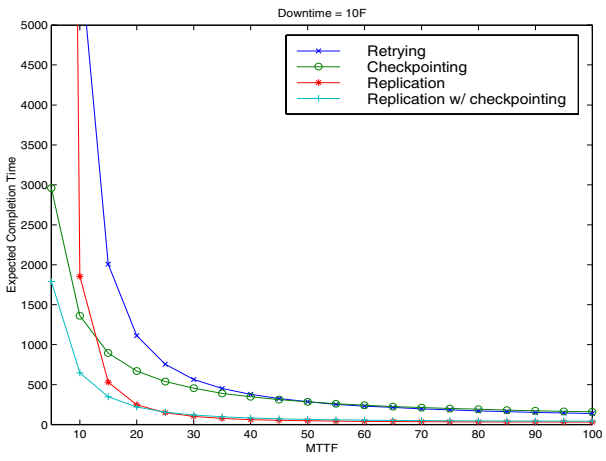

Figure 12. Comparison of expected completion time between different fault tolerance techniques as a function of MTTF when downtime is equal to ten times of the task duration.

process with a probability p of *disk full* exception occurrence.

- We assume that there are no other failures occurring except for the *disk full* exception for the FU, and the SR never fails.

Figure 13 illustrates the importance of supporting exception handling mechanisms. As p becomes close to 1, the expected completion time of the DAG increases exponentially if the FU adopts *masking fault tolerance* techniques such as retrying and checkpointing. In case of $p = 1$, without support for exception handling mechanisms, the execution would never finish successfully.

In conclusion, the results of our experiments in this paper indicate that employing an appropriate failure recovery technique among alternatives depending on the task context and underlying execution environments is critical in reducing the expected completion time. We believe this evaluation validates Grid-WFS's support for diverse failure recovery techniques in highly heterogeneous execution environments like the Grid.

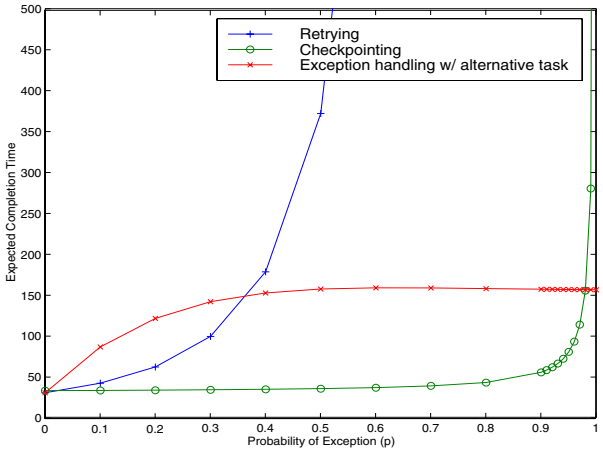

Figure 13. Comparison of the expected completion time as the function of p with retrying, checkpointing, and exception handling w/ alternative task

9 Related Work

Much research has been done on fault tolerance mechanisms in distributed, parallel, and Grid systems. The main focus of that work is on the provision of a single failure recovery mechanism targeting their system-specific domains. Table 1 summarizes fault tolerance mechanisms incorporated in some of traditional distributed systems (e.g., OLTP [14], Ficus [24]), parallel systems (e.g., PVM [12], DOME [3]), Grid systems (Netsolve [5], Mentat [15], Condor-G [10], CoG Kits [26]), illustrating types of failures that they can detect, their failure detection mechanisms and their failure recovery mechanisms. Basically, Our work differs from these systems by its focus on the provision a special form of failure recovery mechanism targeting the *generic, heterogeneous, and dynamic* Grid environments. As can be seen in the Table, none of the systems address the Grid-unique failure recovery requirements mentioned in section 2; no systems support user-defined exceptions; none of them support diverse failure recovery mechanisms, but they provide only a single user-transparent failure recovery mechanism (e.g., *transaction* in OLTP, *checkpointing* in Dome, *retrying* in Netsolve and in Condor-G, *replication* in Mentat).

10 Conclusions

We have argued that the generic, heterogeneous, and dynamic nature of the Grid requires a new form of failure recovery mechanism which should be able to address Grid-unique requirements such as support for diverse failure handling strategies, separation of failure handling strategies from application codes, and user-defined exception handling.

We have designed and prototyped a Grid Workflow System (Grid-WFS), a flexible failure handling framework for the Grid, which meets these Grid-unique failure recovery requirements. We have shown that the framework allows users not only to define failures based on the task context (i.e., user-defined exceptions), but also to specify diverse failure handling strategies using high-level workflow structure which is separated from the application algorithm code. By this separation, we believe, our framework provides a flexible Grid programming paradigm. In the traditional paradigm, failure handling policies are hard-coded in the application code so that if the polices are changed, the corresponding application code should be modified, recomplied, linked, and tested. In this new paradigm, failure handling polices are directly modified by simply changing the corresponding workflow structure.

We have also demonstrated through our experiments that in heterogeneous computing environments like the Grid, it appears to be essential to support multiple fault tolerance techniques and user-defined exception handling in order to achieve high performance as well as fault tolerance in the presence of failures.

Acknowledgements

We would like to thank Ann Chervenak, Karl Czajkowski and anonymous reviewers for providing useful comments on this paper.

References

[1] Condor manuals. http://www.cs.wisc.edu/condor/manual/.

[2] The globus toolkit. http://www.globus.org.

[3] A. Beguelin, E. Seligman, and P. Stephan. Application level fault tolerance in heterogeneous networks of workstations. *Journal of Parallel and Distributed Computing on Workstation Clusters and Networked-based Computing*, June 1997.

[4] S. Brunett, K. Czajkowski, S. Fitzgerald, I. Foster, A. Johnson, C. Kesselman, J. Leigh, and S. Tuecke. Application experiences with the globus toolkit. In *Proceedings of the Eighth IEEE Symposium on High Performance Distributed Computing*, 1998.

[5] H. Casanova, J. Dongarra, C. Johnson, and M. Miller. Application-specific tools. In I. Foster and C. Kesselman, editors, *The GRID: Blueprint for a New computing Infrastructure, Chapter 7.*, pages 159–180. Morgan Kaufmann, 1998.

[6] K. Czajkowski, I. Foster, N. Karonis, C. Kesselman, S. Martin, W. Smith, and S. Tuecke. A resource management architecture for metacomputing systems. In *Proceedings of*

Systems	Failures detected	Failure detection mechanism	Failure recovery mechanism	General comment
Transaction system (e.g., OLTP)	- Host crash - Network failure - Task crash	System-specific polling & event notification	*Transation* (i.e., abort and retry)	Uniform task (i.e., mainly read/write operation)
Distributed file system (e.g., Ficus)	- Host crash - Network failure	Voting	*Replication*	Uniform task
PVM	- Host crash - Network failure - Task crash	System-specific polling & event notification	Diverse failure handling in the application context	Should hardcode recover strategies in the application
DOME	- Host crash - Network failure - Task crash	System-specific polling & event notification	*Checkpointing*	Targeting the SPMD parallel applications
Netsolve	- Host crash - Network failure - Task crash	Generic heartbeat mechanism	*Retry* on another available machines	Grid RPC
Mentat	- Host crash - Network failure	Polling	*Replication*	Exploiting task's stateless and idempotent nature
Condor-G	- Host crash - Network crash	Polling	*Retry* on the same machine	Use of Condor client interfaces on top of Globus
CoG Kits	N/A	N/A	N/A	Should hardcode failure detection (e.g., timeout) and recovery strategies

Table 1. Fault tolerance mechanisms in traditional distributed & parallel, and Grid systems

the IPPS/SPDP '98 Workshop on Job Scheduling Strategies for Parallel Processing, 1998.

[7] A. Duda. The effects of checkpointing on program execution time. *Information Processing Letters*, 16:221–229, June 1983.

[8] M. C. Elder. *Fault Tolerance in Critical Information Systems*. PhD thesis, University of Virginia, 2001.

[9] I. Foster and C. Kesselman, editors. *The GRID: Blueprint for a New computing Infrastructure*. Morgan Kaufmann, 1998.

[10] J. Frey, T. Tannenbaum, I. Foster, M. Livny, and S. Tuecke. Condor-g: A computation management agent for multi-institutional grids. *Cluster Computing*, 5(3), 2002.

[11] F. C. Gartner. Fundamentals of fault-tolerant distributed computing in asynchronous environments. *ACM Computing Surveys*, 31(1), Mar. 1999.

[12] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, B. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine: A User's Guide and Tutorial for Network Parallel Computing*. MIT Press, 1994.

[13] D. Georgakopoulos, M. Hornick, and A. Sheth. An overview of workflow management: From process modeling to workflow automation infrastructure. *Distributed and Parallel Databases*, 3(2):119–153, Apr. 1995.

[14] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, 1994.

[15] A. S. Grimshaw, A. Ferrari, and E. A. West. Mentat. In G. V. Wilson and P. Lu, editors, *Parallel Programming Using C++, Chapter 10*, pages 382–427. The MIT Press, Cambridge Mass., 1996.

[16] D. Hollingsworth. Workflow management coalition the workflow reference model. WfMC-TC00-1003, Nov. 1994.

[17] S. Hwang. *Grid Workflow: A Flexible Framework for Fault Tolerance in the Computational Grid*. PhD thesis, University of Southern California, 2003. To appear.

[18] S. Hwang and C. Kesselman. A generic failure detection service for the grid. Technical Report ISI-TR-568, USC Information Sciences Institute, Feb. 2003.

[19] L. Kleinrock. *Queueing Systems, Volume 1: THEORY*. Wiley-Interscience Publication, 1975.

[20] N. Krishnakumar and A. Sheth. Managing heterogeneous multi-system tasks to support enterprise-wide operations. *Distributed and Parallel Databases*, 3(2):155–186, Apr. 1995.

[21] J. Leon, A. L. Fisher, and P. Steenkiste. Fail-safe pvm: A portable package for distributed programming with transparent recovery. Technical Report CMU-CS-93-124, Carnegie Mellon University, Feb. 1993.

[22] J. S. Plank, M. Beck, G. Kingsley, and K. Li. Libckpt: Transparent checkpointing under unix. In *Proceedings of the the USENIX Winter Technical Conference*, New Orleans, Louisiana, Jan. 1995.

[23] J. S. Plank and W. R. Elwasif. Experimental assessment of workstation failures and their impact on checkpointing systems. In *In Proceedings of the 28th Fault-tolerant Computing Symposium (FTCS-28)*, June 1998.

[24] G. J. Popek, R. G. Guy, T. W. Page, Jr., and J. S. Heidemann. Replication in Ficus distributed file systems. In *IEEE Computer Society Technical Committee on Operating Systems and Application Environments Newsletter*, volume 4, pages 24–29. IEEE Computer Society, 1990.

[25] G. Stellner. Cocheck: Checkpointing and process migration for mpi. In *In 10th International Parallel Processing Symposium*, pages 526–531. IEEE Computer Society, Apr. 1996.

[26] G. von Laszewski, I. Foster, J. Gawor, W. Smith, and S. Tuecke. Cog kits: A bridge between commodity distributed computing and high-performance grids. In *ACM 2000 Java Grande Conference*, 2000.