Implementation of the fault tolerance in computational grid using agents by meta-modelling approach

C. Srimathi* and J. Vaideeswaran

School of Computing Science and Engineering,
VIT University, Vellore, 632014, Tamil Nadu, India
E-mail: csrimathi@vit.ac.in
E-mail: jvaideeswaran@vit.ac.in
*Corresponding author

Abstract: In the grid environment, some of the nodes may be working while others may not be active. Alternatively, all the computers could be operational, but their interconnection network may fail. From the perspective of one computer, such network partitioning may appear as a failure to other computers. These types of failures may lead to a major impact on the whole application, which is executing on the Grid for many days. In this paper we will be meta-modelling the computational grid and implementing the fault tolerant mechanism using Java agents. The purpose of the work proposed in this paper is to automate the development of a computational grid and creating graphical workflows of applications using domain-specific modelling techniques. This paper is to provide a high level view for the construction of Grid applications with the flexibility in design and deployment.

Keywords: agents; scheduler; replication; check-pointing; computational grid.

Reference to this paper should be made as follows: Srimathi, C. and Vaideeswaran, J. (xxxx) 'Implementation of the fault tolerance in computational grid using agents by meta-modelling approach', *Int. J. Communication Networks and Distributed Systems*, Vol. X, No. Y, pp.000–000.

Biographical notes: C. Srimathi is currently working as an Assistant Professor (Selection Grade) in School of Computing Science and Engineering. She received his BE in Electrical and Electronics from Bharathiar University, India and his ME (CSE) from Madurai Kamaraj University, India. Her research interests include grid computing and agent-based computing. She is currently working on a funding project sanctioned by a government agency in India.

J. Vaideeswaran is working as a Senior Professor in School of Computing Science and Engineering. He received his PhD from Anna University, Chennai. He has several national and international publications. His research interest include software engineering, computer architecture, grid computing and embedded systems.

1 Introduction

Grid computing emerges as a distributed infrastructure for large-scale data processing and scientific computing. One of the objectives of computational grids is to offer applications the collective computational power of distributed but typically shared heterogeneous resources. Unfortunately, efficiently harnessing the performance potential of such systems (i.e., how and where applications should execute on the grid) and successfully executing the applications without any failures are the challenges principally to the distributed, shared and heterogeneous nature of the resources involved. The essence of a Grid can be represented by a checklist proposed by Foster (2006):

1 a grid coordinates resources that are not subject to centralised control

2 a grid uses open, standard protocols and interfaces

3 a grid is able to deliver non-trivial qualities of service.

This checklist acts as an informal definition of 'grid' and a system that is called a grid must fulfil these requirements. Grid applications require lot of computation power and will perform long tasks which may run for many days or months on different boundaries and heterogeneous nodes require powerful mechanism to handle failures (Naveed, 2006). Failures may occur due to various reasons such as dynamic nature, heterogeneous nodes, network QoS, accidentally user shutdowns node and physical damage. These are some of the problems arises in grid environments. Detecting the root cause of these problems in a massive environment is very difficult (Schmidt, 2003). Due to the diverse faults and failure conditions, developing, deploying and executing long running applications over the grid remains a challenge. So fault tolerance is an essential factor for grid computing.

Fault tolerance (Siva Sathya et al., 2007) is the ability to preserve the delivery of expected services despite the presence of fault caused errors within the system itself. It aims at the avoidance of failures in the presence of faults. A fault tolerant service detects errors and recovers from them without participation of any external agents, such as humans.

This paper focuses on modelling the computational grid, interpret them and add agents to the existing workflow for fault diagnosis and make the precautionary measure. The exact problem which we are considering in our paper is partial failures – the failure of apart, but not all, of the system. The grid computing infrastructure is designed and implemented using meta-modelling approach, which is mainly concerned with fault tolerance by using job replication. We mainly considered three types of failures node operating system and network. We will first model the grid computing architecture using GME and identify the fault prone areas and apply the necessary mechanism like check-pointing and mostly replication mechanism wherever necessary.

When designing fault tolerant systems, we have assumed that hardware or software components may fail. Faults in embedded software in computational Grid are commonly caused by deviations from specification of the supporting platforms or programming errors. Fault tolerance is a technique that attempts to neutralise the potential faults to avoid system failures, by incorporating redundancy in system components. Blindly duplicating all components that may fail in the implementation platform is clearly an inefficient solution in a computational grid. By monitoring the node status using agents for each node we can load the executing application to another available node in the grid

environment. In this way we can identify node failure and can do recovery process by migrating application transparently to another node.

When all the nodes in the grid environment are busy, then agent will fail to locate the available node with the suitable resources to execute the application. We assumed that there are no application or process failures when a application is loaded to get executed. Because processes usually fail by two reasons:

1 *Implementation errors (bugs):* are caused by programming errors, such as a programme tries to access an absent vector position. Since the process cannot be recovered, the error has to return and taken care.

2 *Incidental errors (transient bugs):* are typically related to execution environment conditions, such as hardware fault (e.g., transient devices faults), limit conditions (e.g., out of storage/memory, counter overflow), race conditions.

We will be designing a agent-based approach to migrate the job which is executing on a node (going to failure) to another node which is available in the grid by using the fault tolerant workflow designed for computational grid in generic modelling environment (GME) which gives automatic code generation.

The paper is organised in the following way. Section 2 deals about the related work in the computational grid. The proposed system architecture with the implementation are given in Sections 3 and 4 and conclusion and future work in Section 5.

2 Related work

There are lot more grid projects implemented today, with different objectives, implementation issues, target applications and computer infrastructure. Each of them has a particular manner to treat the occurrence of failures.

A fault tolerance framework for grids is presented in Hwang and Kesselman (2003). It consists of failure detection service (FDS) and a failure handle service (Grid-WFS). The FDS enables the detection of both application crashes and user-defined exceptions. The Grid-WFS allows users to achieve failure recovery in a variety of ways, depending on requirements and constraints of their applications. It uses a workflow structure as a high-level recovery policy specification, which enables support for multiple failure recovery techniques and separating failure handling strategies from application code. FDS requires that the user explicitly registers the application in a heartbeat monitor, through an application programming interface (API) (Foster, 2005). Condor provides fault tolerance for grid applications through check pointing approach. To be executed with check-pointing support in the condor system, grid applications must be re-linked (but not re-compiled) to include the condor checkpoint library. This library allows condor fault tolerance mechanism to periodically capture the application checkpoint, and gives to programmes the ability to checkpoint itself at any moment. These checkpoints are stored with compression in a checkpoint server, i.e., a dedicated machine that stores all system checkpoints.

Traditionally, software dependability has been achieved by fault avoidance techniques such as structured programming and software reuse in order to prevent faults, or fault removal techniques such as testing in order to detect and delete faults. However, in the case of grid computing, these approaches, although still very much useful, may not

be enough (Mincer-Daszkiewicz, 2006). Fault tolerance is an important property in grid computing as the dependability of individual grid resources may not be guaranteed. As resources are used outside of organisational boundaries, it becomes increasingly difficult to guarantee that a resource being used is not malicious in some way. In many cases, an organisation may send out jobs for remote execution on machines upon which trust cannot be placed; for example, the machines may be outside of its organisational boundaries, or may be desktop machines that many people have access to.

A fault tolerant approach may therefore be useful in order to potentially prevent a malicious node affecting the overall performance of the application. As applications scale to take advantage of grid resources, their size and complexity will increase dramatically. However, experiences (Abawajy, 2004) tell that systems with complex asynchronous and interacting activities are very much prone to errors and failures due to their extreme complexity, and we simply cannot expect such applications to be fault free, no matter how much effort is invested in fault avoidance and fault removal. In fact, the likelihood of errors occurring may be exacerbated by the fact that many grid applications will perform long tasks that may require several days of computation (de Camargo et al., 2004). Hence, the cost and difficulty of recovering from faults in grid applications is higher than that of normal applications. Furthermore, the heterogeneous nature of grid nodes means that many grid applications will be functioning in environments where interaction faults are more likely to occur between disparate grid nodes. The heterogeneous architectures and operating system platforms (Fukuda and Smith, 2006), give rise to a number of problems that are not present in the traditional homogeneous systems.

The complexity of both

a varying architectural features, such as data representation and instruction sets

b varying operating system features, such as process management and communication interfaces, must be masked from the application programmer.

If no fault tolerance is provided, the system cannot survive to continue when one or several processes fail, and the whole programme crashes. In this sense, a technique is needed that would enable a system to perform fault tolerant procedures that can continue to execute even in the presence of a fault. Since the primary purpose of grids is to provide computational power to scientists, the intended application developer is a domain expert, typically a physicist, a biologist or a meteorologist, not necessarily a grid expert (Gupta et al., 2001). So, most application developers are unaware of the different types of failures that may occur in the grid environment and hence, fault tolerance becomes increasingly important. Our paper mainly concentrates on the identifying the architectural features and the operating system features and failures involved in it and provide the precautionary measure. This meta-modelling approach mainly concentrates on identifying the root cause of the failure which is the biggest challenge in the heterogeneous grid environment.

3 Proposed system

The proposed system focuses on three research issues such as:

1 The creation of a meta-model that maps the computational grid components to a graphical model. This meta-model defines the language used to construct workflow models.

2 The generation of Java programmes from the graphical workflows. This is realised by using a model interpreter that traverses the graphical workflows and generates a programme that manages the application execution using GME.

3 We will be introducing agents in the Java programmes which has been generated in the fault prone areas and making the precautionary measure before actually deploying in the grid computing environment.

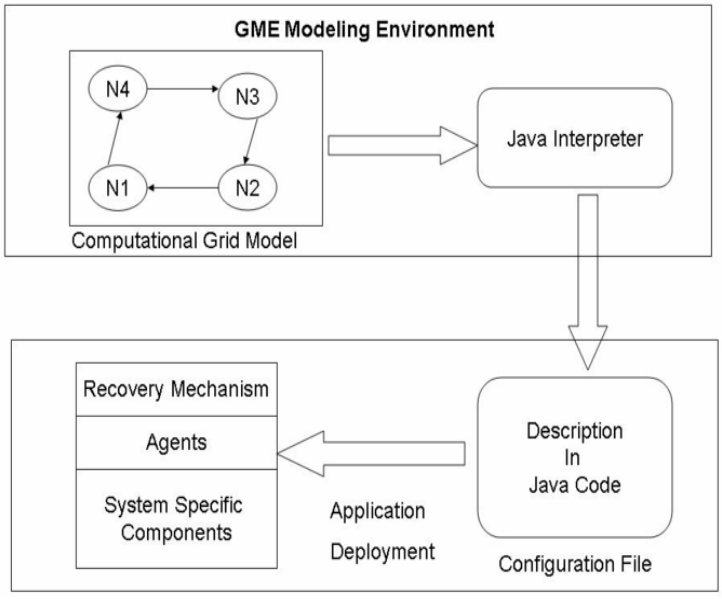
Two actions are necessary to create domain-specific models for the computational grid

1 definition of the meta-model and defining the paradigm (language) to be used to create workflow models

2 implementation of the interpreter that translates the workflow models into corresponding Java code.

Both of these actions are implemented using GME. One of the advantages of using GME is that it allows a modeller to define base elements that can be reused in more complex models as shown in Figure 1.

Figure 1 Proposed system architecture



Model integrated computing (MIC) facilitates model analysis and automatic programme synthesis by incorporating model integrated programme synthesis (MIPS) to transform a model in a specific domain to a physical artefact. More precisely, GME is a domain specific modelling tool that supports the MIC methodology and contains a MIPS environment for interpreting and generating physical artefacts of that model. GME uses the unified modelling language (UML) and the object constraint language (OCL) technologies to construct domain specific environments.

GME uses UML class diagrams to compose domain specific environments. GME supports MIPS through the Builder Object Network version 2.0 (BON2). BON2 is a GME component interface used to transform models to physical artefacts. BON2 consists of software classes and interfaces to support the interpretation of a model. These classes and interfaces are automatically generated by GME, and most of them are independent of the specific domain (Sharma and Bawa, 2008). However, a domain specific interface is generated that allows traversal of objects in a specific domain.

In a grid environment, workflow execution failure can occur for various reasons: the variation in the execution environment configuration, non-availability of required services or software components, overloaded resource conditions, system running out of memory, and faults in computational and network fabric components. Grid workflow management systems should be able to identify and handle failures and support reliable execution in the presence of concurrency and failures (Foster et al., 2004). So failure prone areas are identified and agents are deployed in that areas .We will be recovering from the failure by using the following:

- *retrying* – stopped and started from scratch
- *check-pointing* – repository the execution states and restores at point of failure
- *replication* – simultaneously executing task on n different nodes
- *check-pointing and replication* – this mechanism is highly required for high-end tasks.

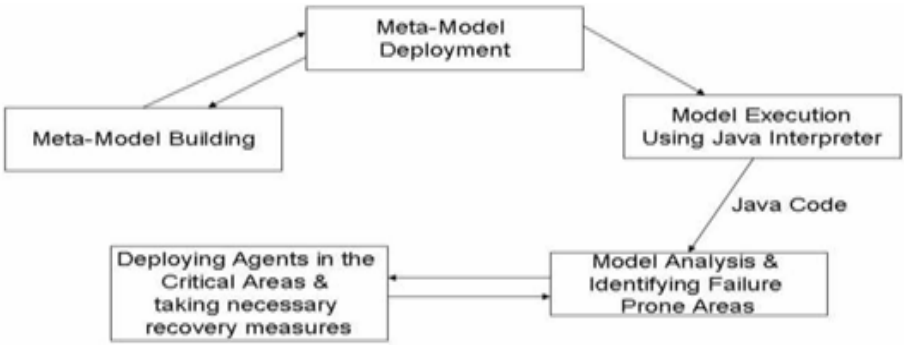
The overall design flow of a modelling environment for computational grid applications is presented in Figure 1. The flow begins with the designer constructing a visual application model for the computational grid paradigm. GME stores objects (nodes in the grid computing) in the model in a model database. The designer instantiates an interpreter from the GME user interface to initiate the model interpretation on objects in the model database. The graphical model is interpreted to generate a configuration file for the execution model (the run-time environment). The configuration file is used to generate and build the code for a complete executable model of the computational grid graph.

3.1 The interactive meta-modelling workflow system

In domain-specific modelling, a design engineer creates models for a specific domain using concepts and terminology from that domain (Bronevetsky et al., 2006). The domain specific models are developed by first creating a meta-model that specifies the ontology of the domain. The meta-model serves as a paradigm, or language, that defines the syntax and static semantics for models of that domain; the dynamic semantics are introduced by an interpreter that synthesises the models into different representations (Gioiosa et al.,

2005) (e.g., XML configuration files or source code). The GME is a graphical tool that automates the creation of domain-specific models. The interactive work flow meta-modelling consists of five stages and the process involved in the system design is shown in Figure 2:

Figure 2 Application development process flow



3.1.1 Model building

The step includes not only the software, but also a precise specification of the functionality it provides with all dependencies (hardware, software, version and other components .We model the complex system so that it helps in understanding and also simplifies the testing and maintenance of the computational grid.

The model-driven architecture (MDA) has been proposed to promote the concept of a common stable meta-model, which is language-, platform- and vendor-neutral. The main idea is that even if the underlying infrastructure shifts over time, the meta-model remains stable, and the only changes are the tools to bridge the meta-model and infrastructure. Computational grid usually consists of number of computing elements. Each computing element consists of the hardware and software characteristics.

The resource description of the computational grid consists of hardware and a software design. Figure 3 explains the meta-modelling characteristics involved in the hardware part of the computing element. The resources are scored based on the ranking. The ranking is done based on the processor, memory configurations and MTTF (mean time to failure). High priority jobs which are failed are given to the lower ranked resource. The meta-modelling design for the software of the computing element is also done similarly. The hardware and software design is done mainly to categorise the fault.

Design of the software component of the computational grid that deals with meta-modelling is shown in Figure 4. By the design of the above two components we will be able to differentiate the software and hardware faults.

The root cause of the failure can be clearly identified in the case of the hardware faults. The cluster design is constructed by the taking into account the above components. Figure 5 shows the meta-model construction of the cluster. We have taken to be cluster as we are grouping the resources in the computational grid by the characteristics of the hardware and software. Homogeneous resources are grouped into single cluster so that fault tolerance can be achieved to a certain extent especially at the hardware level.

Figure 3 Meta-modelling design for hardware of the computing element

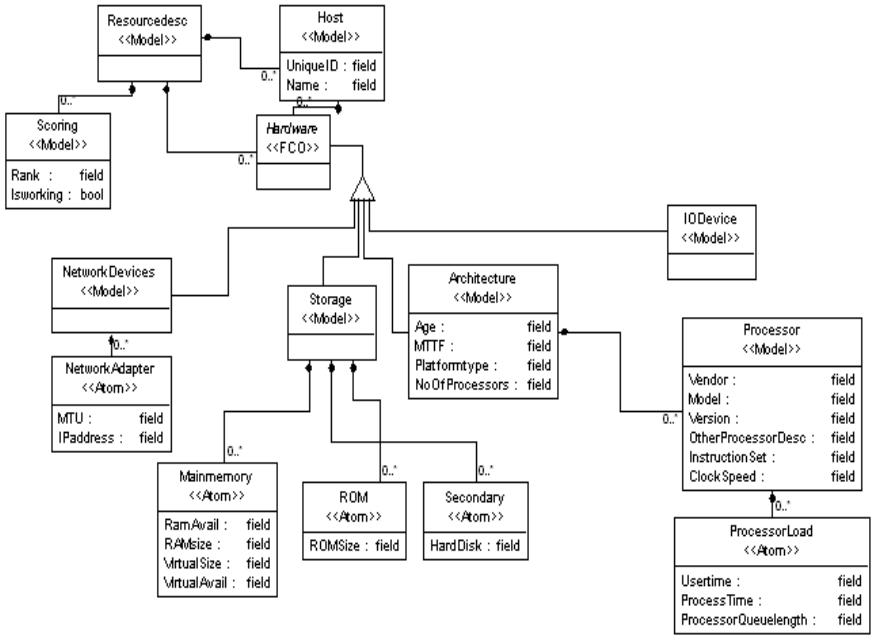


Figure 4 Meta-modelling design for software of the computing element

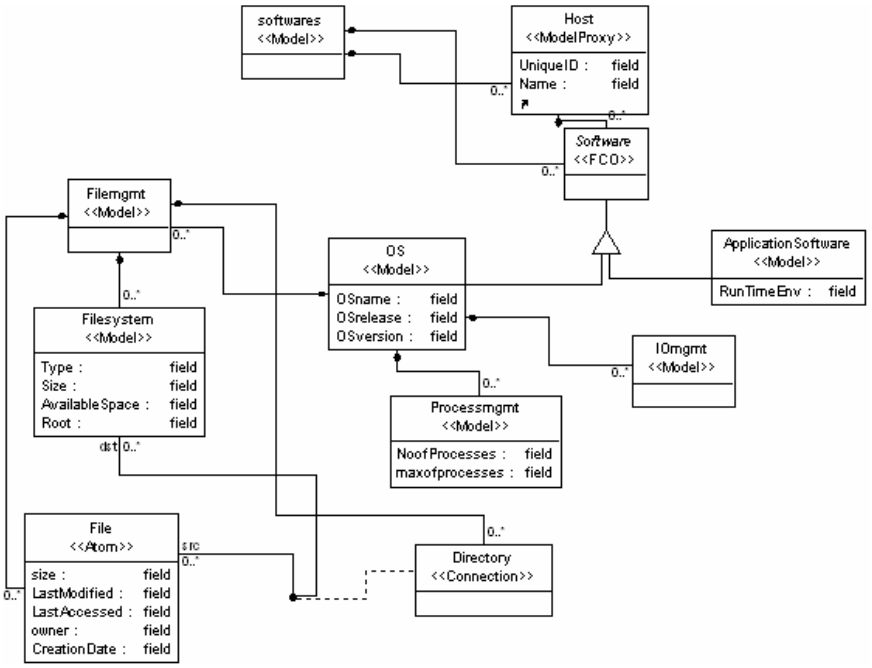
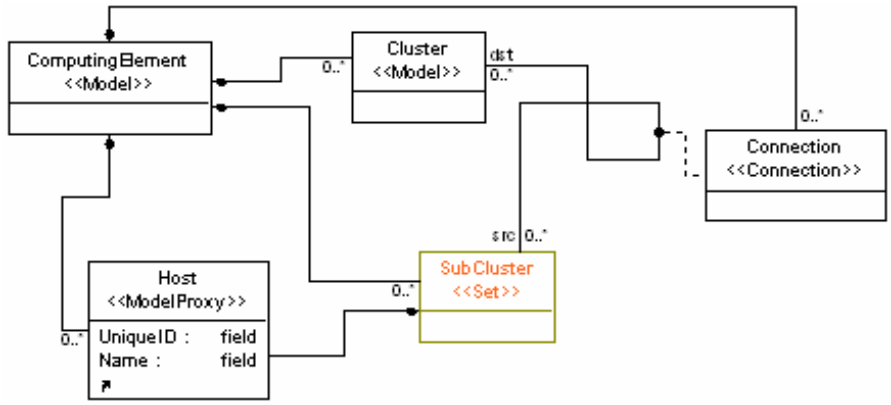


Figure 5 Meta-modelling design for cluster of the computational grid (see online version for colours)



3.1.2 Model deployment and analysis

Operations are the smallest element in the grid workflow. Based on the 'problem description' of the operation in the workflow, it picks up appropriate and available grid services, schedules the operations to the grid resources, execute the workflow operation, and collects the operation results. Then, operations are executed on the grid resources within the organisations. Each operation in a grid workflow corresponds to a computational operation and is usually carried out on an individual grid node.

Figure 6 Modelling of the cluster after interpretation (see online version for colours)

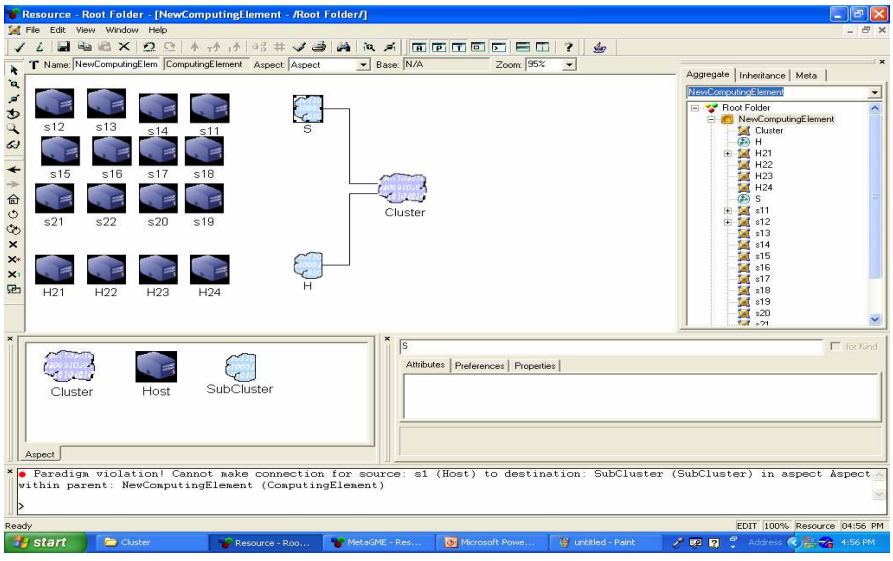
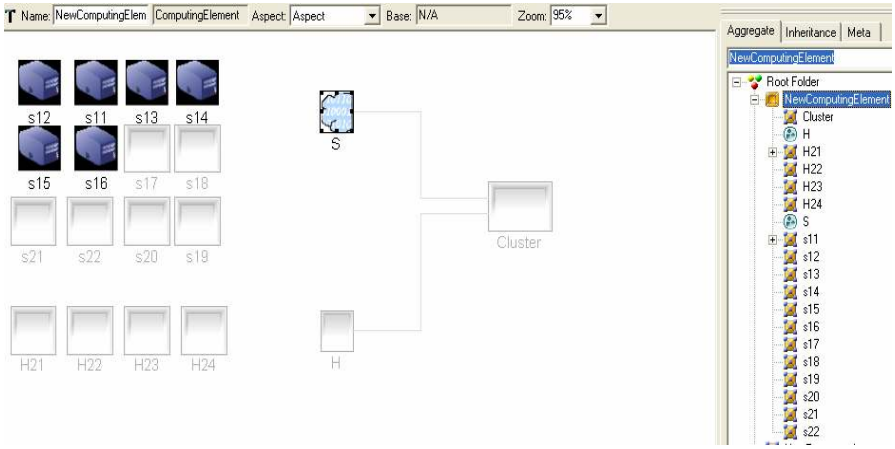


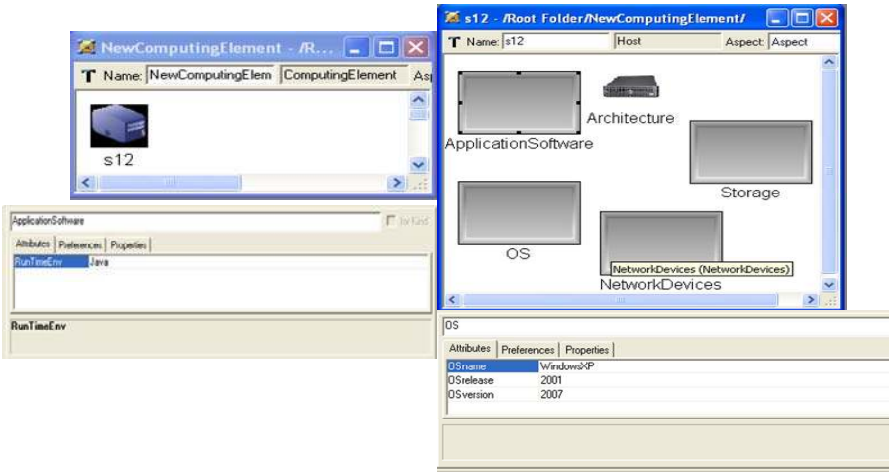
Figure 7 Resources are sorted to the ranks (see online version for colours)



The meta-modelling design is interpreted and 16 nodes are taken. In that 12 nodes are grouped together into the single cluster S and the other four nodes are put in the separate cluster H (Figure 6). Each resource consists of both hardware and software characteristics. The active nodes are highlighted (Figure 7) where the resources are ready to take up the jobs.

The resources that are highlighted are ready to take up the jobs and the jobs will be allotted based on first come first serve basis. Resources are sorted in the descending order according to the ranks. The individual computing element analysis is given in Figure 8.

Figure 8 Analysis of a computing element in the computational grid (see online version for colours)



3.1.3 Model analysis and identifying the failure prone areas

The failure prone areas are identified using the producer and consumer agents. The broker agent plays a very vital role in achieving the above mentioned task. The joining entity here is a virtual organisation of the grid and objects refer to the nodes in the VO's of the computational grid. A joining entity comprises one or more cooperating objects in the Java programming language that have just received a reference to the lookup service and are in the process of obtaining services from, and possibly exporting them to a node. An entity may be a discovery entity, a joining entity, or an entity that is already a member (Medeiros et al., 2003). A group is a logical name by which a Java node is identified. In its most basic form, a consumer decomposes a job into discrete tasks with a unique Agent ID (Figure 9). Each task represents one unit of work that may be performed in parallel with other units of work. Tasks are associated with objects written in the Java programming language 'Java objects' that can encapsulate both data and executable code required to complete the task. The consumer writes the tasks into a space, and asks to be notified when the task results are ready. Producer query the space to locate tasks that need to be worked on and get listed the Agent ID. Each producer takes one task at a time from the space and performs the task computation. When a producer completes a task, it writes a result back into the space, and attempts to take another task. The consumer side agents takes the results from the space, and reassembles them, if necessary, to complete the job.

Whenever a consumer (user) wants to execute a job, he will put the job into broker (space) subdivided into small tasks. Like a service registration into Lookup registrar. The producers which get notification and see into space will take the tasks and execute them. Number of producers is the strength of the grid and its computation power. After execution completes, the results are written to space and if some more tasks are there, again producer will take and execute. The broker is a rich shared memory location which is capable of handling tasks and executed results. It will notify the results back to the consumer.

The consumers perform computing tasks by first locating then using suitable services. A central issue in this system is how to find services. Java provides the lookup services typically by service type and/or service attributes. In this current system, when ever a service is registered in the LUS with a unique serviceID, a new agent is deployed with a unique AgentID (ID is created by combination of host address and system current time in milliseconds). So no other agent can have the same ID. The lookup service uses the required service interface and attributes during service matching in its lookup ( ) method. When connecting to the system, producers also need to locate lookup services, and then find brokers. By tracking the AgentID we will be easily find out the failure prone areas.

3.1.4 Deploying agent and recovery mechanisms

Computing processes may be thought of as a kind of multi-agent cooperation, in the sense that an individual or a group of grid workflow agents can be used to perform an operation in a workflow, and a workflow can be used to orchestrate or control the interactions between grid services or agents. Multiple grid services or agents working cooperatively may accomplish a particular part of the workflow process, such as fault tolerant issue.

For recovery mechanisms, we are planning to apply state or code migration. These refer code transfer or state transfer of the agent. The state migration is composed of more aspects. The state transfer of an agent consists of current execution point and the current data of the agent.

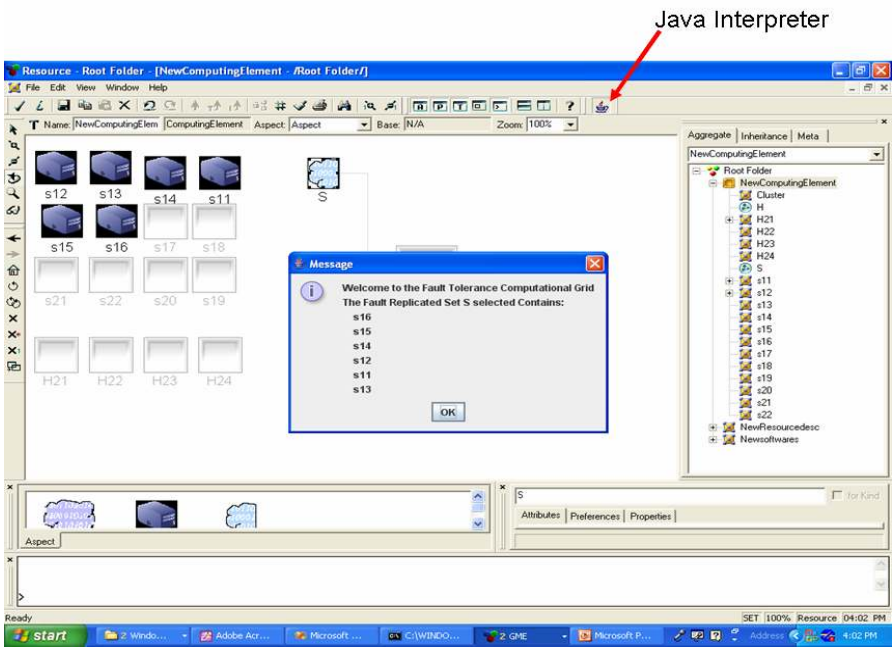
Figure 9 Proposed pseudo code for creation of sample agent

```
//Assigning a Unique random number for a creation of agent process  which is static
Public static final long serialVersionUID = 4057062707324824434L;
// Generating the unique Agent identifier
public AgentIdentity() {
if (ID == null) {
originatorIP = java.net.InetAddress.getLocalHost().getHostAddress();
creationTime = System.currentTimeMillis();
ID = UuidFactory.generate();
// A key will be generated and assigned
}}
// comparison of two agent ID's for finding out either same agent or another agent-
AgentID Should be unique
public boolean equals(Object o) {
if (o instanceof AgentIdentity) {
AgentIdentity cmp = (AgentIdentity) o;
if (cmp.getOriginatorIP().equals(originatorIP)) {
if (cmp.getCreationTime() == creationTime) {
if (cmp.getID().equals(ID)) {
return true;
}}}}
return false;}
// splitting ID for comparision
public AgentIdentity(String ID) {
String[] parts = ID.split("\\|");
System.out.println(parts.length);
if (parts.length != 3)
throw new RuntimeException("AgentIdentity not in correct format");
originatorIP = parts[0];
creationTime = Long.parseLong(parts[1]);
this.ID = UuidFactory.create(parts[2]);
}}
// finding which machine has deployed the agent
public String getOriginatorIP() {
return originatorIP;
}
public int hashCode() {
return originatorIP.hashCode() + ID.hashCode();
}
```

3.1.5 Model execution using Java interpreter

After the workflow is specified, a model interpreter traverses the internal representation of the model and generates the control code that manages the application execution. The interpreter first gathers all the information from resources, jobs and other details from schedulers.

Figure 10 Final Java interpretation (see online version for colours)



This information, along with the specification of the application workflow, constitutes the interpreter's input. The interpreter then executes the semantic actions associated with each workflow task. The output of this step is a Java programme that manages the application execution is shown in Figure 10. The job replication mechanism algorithm for fault tolerance and match making algorithm is used for scheduling.

4 Experimental studies

This research work aims at designing the computational grid environment in GME and deploy agent that will aid in automatic scheduling of jobs to the computer resources with the help of scheduler and takes care for fault tolerance. The approach uses a visual modelling environment where the designer can enter models and realise application instances from the computational grid such that errors can be easily reduced and the time to analyse the root cause of the problem is decreased to a greater extent.

For doing the analysis a particular experiment was set up and we hypothesised that an agent oriented fault-tolerant approach will be more efficient if the time taken for interpreting the GME application in a computational grid environment is minimum. The tasks are performed only once and can be reused for any subsequent application. The estimated time to develop and create the grid environment in GME assumes a basic understanding of UML class diagrams.

Table 1 lists the various test cases for the experimental setup. Here we have assumed that same number of agents was used for injecting and detecting the faults. The faults were injected randomly in any of the cluster. Types of faults and its recovery mechanism

have been identified for injecting into the computational grid through agents is given in Table 2.

Table 1 Test case for analysis

No of nodes	*1st cluster*	*2nd cluster*
16 (case 1)	12	4
32 (case 2)	24	8
128 (case 3)	96	32
1,024 (case 4)	768	256

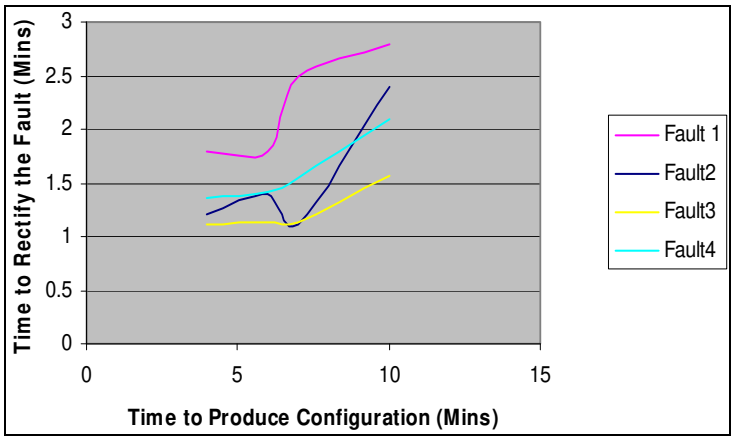
Table 2 Types of fault and recovery mechanism

S. No	*Fault type*	*Recovery mechanism*
1	Network level (packet loss)	Retrying
2	Operating system fault	Check-pointing
3	Memory level (memory leak	Replication
4	Software fault (unknown exception)	Check-pointing and replication

Table 3 Implementation time of the design environment

Task	*Time in minutes*
Meta-modelling the computational grid	45
Model deployment	12
Injecting the fault through agents-agent development	90
Final result – set identification and generation of interpreter code	108

Figure 11 Analysis of the various faults with different test cases with other parameters fixed (see online version for colours)



A summary of the actual time and effort to create the computational grid design, agent deployment and run time environment (Table 3). Each fault was injected in all the four test cases, totally 16 sets of analysis were done. Time to inject the Fault 1, Fault 2,

Fault 3 and Fault 4 through the agents were eight, nine, ten and ten respectively and constant for all the test cases. Analysis (Figure 11) shows that even though the no of nodes were increased from 16 to 1,024, time to rectify the faults was not much varying . The most serious fault was the network fault (packet loss) which has to be started from scratch. For other type of faults there was not much variation in the time for detecting the type of failures in all the four test cases.

5 Conclusions

In this paper, we proposed and implemented approach for modelling the grid computing architecture and analysing the failure tolerant and mechanism to recover from the failures that are hardware oriented. When exploring various workflows scenarios in a complex environment like computational grid, modelling tools and their interpreters facilitate the more rapid ability to change the workflow details. That is, it is easier to manipulate and change domain models rather than the associated code and it is easier to find out the root cause of the problem. The advantage of this approach is that we can drill down to the lower levels of the computational grid architecture.

Model-driven techniques possess the ability to generate multiple artefacts from the same model. Thus, with the same domain knowledge different output representations can be generated. Domain modelling removes the accidental complexities of creating workflows in a grid by focusing on higher levels of abstraction at the problem space rather than solution space.

The current trend in grid computing is moving towards service oriented architecture. To make the environment capable of moving in that direction, future work will be focused in two aspects:

1 the utilisation of grid services as workflow tasks

2 the capability of generating grid services from workflows.

References

Abawajy, J.H. (2004) 'Fault-tolerant scheduling policy for grid computing systems', *Proceedings of the 18th IEEE International Parallel and Distributed Processing Symposium*, 26–30 April, pp.342–345, Santa Fe, New Mexico, North America.

Bronevetsky, G., Fernandes, R., Marques, D., Pingali, K. and Stodghill, P. (2006) 'Recent advances in check-point/recovery systems', *Proceedings of IEEE International Parallel & Distributed Processing Symposium*, 25–29 April, pp.245–249, Rhodes Island, Greece.

de Camargo, R.Y., Goldchleger, A., Kon, F. and Goldman, A. (2004) 'Checkpointing-based rollback recovery for parallel applications on the InteGrade grid middleware', *2nd Workshop on Middleware for Grid Computing*, 18–22 October, Toronto, Canada.

Foster, I. (2005) 'Globus toolkit version 4: software for service-oriented systems', in the *Proceedings of IFIP International Conference on Network and Parallel Computing*, LNCS 3779, 30 November to 3 December, pp.2–13, Beijing, China.

Foster, I. (2006) 'What is the grid? A three point checklist', 20 July 2002, available at http://www-fp.mcs.anl.gov/~foster/Articles/WhatIsTheGrid.pdf (accessed on 7 July 2011).

Foster, I., Jennings, N.R., Kesselman, C. (2004) 'Brain meets brawn: why grid and agents need each other', *AAMAS'04*, ACM, 19–23 July, New York, USA.

Fukuda, M. and Smith, D. (2006) 'UWAgents: a mobile agent system optimized for grid computing', *Proceedings of the International Conference of Grid Computing and Application GCA 2006*, 26–29 June, Las Vegas, Nevada, USA.

Gioiosa, R., Sancho, J.C., Jiang, S. and Petrini, F. (2005) 'Transparent, incremental checkpointing at Kernel level: a foundation for fault tolerance for parallel computers', *Proceedings of the 2005 ACM/IEEE on Superccomputing-SC'05 Conference*, 12–15 November, pp.45–147, Washington, USA.

GME documentation and software, available at http://www.isis.vanderbilt.edu/Projects/gme/ (accessed on 21 August 2010).

Gupta, T.D., Chandra, G.S. and Goldszmidt, G.S. (2001) 'On scalable and efficient distributed failure detectors', in the proceedings *20th ACM Symposium, On Principles of Distributed Computing*, 26–28 August, Newport, RI, USA.

Hwang, S. and Kesselman, C. (2003) 'A flexible framework for fault tolerance in the grid', *Journal of Grid Computing*, Vol. 3, No. 1, pp.251–272.

Java spaces principles and Jini documentation for programming obtained through the internet, available at http://java.sun.com/docs/books/jini/javaspaces/ (accessed on 15 December 2010).

Medeiros, R., Crine, W., Brasilerio, F. and Saure, J. (2003) 'Faults in grids: why are they so bad and what can be done about it? In grid computing', *Proceedings of Fourth International Workshop On Grid Computing*, 17 November, pp.18–24, Arizona, USA.

Mincer-Daszkiewicz, J. (2006) 'A service for reliable execution of grid applications', a thesis based on which an article with the same title is submitted to the in CoreGRID Workshop on Grid Middleware in Dresden, Germany.

Naveed, A. (2006) 'A planning-based approach to failure recovery in distributed systems', PhD thesis, University of Colorado, Boulder, Colorado, USA.

Schmidt, H. (2003) 'Trustworthy components: compositionality and prediction', in *Journal of Systems and Software*, Vol. 65, No, 3, pp.215–225, Elsevier.

Sharma, A. and Bawa, S. (2008) 'Comparative analysis of resource discovery approaches in grid computing', *Journal of Computers*, Vol. 3, No. 5, pp.60–64.

Siva Sathya, S., Kuppuswami, S. and Syam Babu, K. (2007) 'Fault tolerance by check-pointing mechanisms in grid computing', *Proceedings of the International Conference on Global Software Development*, 15–18 July, pp.243–248, Coimbatore, India.