GEMS: A Fault Tolerant Grid Job Management System

Sriram Satish Tadepalli

Thesis submitted to the faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE
in
Computer Science and Applications

Dr. Calvin J. Ribbens
Dr. Dennis G. Kafura
Dr. Srinidhi Varadarajan

December 19, 2003
Blacksburg, Virginia

Keywords: Grid Computing, Grid Job Management Systems, Local Resource Manager, Job Migration, Fault Tolerance.

Copyright 2003, Sriram Satish Tadepalli

GEMS: A Fault Tolerant Grid Job Management System

by

Sriram Satish Tadepalli

Committee Chairman: Dr. Calvin J. Ribbens

Computer Science and Applications

(ABSTRACT)

The Grid environments are inherently unstable. Resources join and leave the environment without any prior notification. Application fault detection, checkpointing and restart is of foremost importance in the Grid environments. The need for fault tolerance is especially acute for large parallel applications since the failure rate grows with the number of processors and the duration of the computation.

A Grid job management system hides the heterogeneity of the Grid and the complexity of the Grid protocols from the user. The user submits a job to the Grid job management system and it finds the appropriate resource, submits the job and transfers the output files to the user upon job completion. However, current Grid job management systems do not detect application failures.

The goal of this research is to develop a Grid job management system that can efficiently detect application failures. Failed jobs are restarted either on the same resource or the job is migrated to another resource and restarted. The research also aims to identify the role of local resource managers in the fault detection and migration of Grid applications.

TABLE OF CONTENTS

1 Introduction **1**

1.1 Thesis . . . 2

2 Related Work **3**

2.1 Middleware for Executing an Application on the Grid . . . 3

2.1.1 Monitoring and Directory Service . . . 3

2.1.2 Grid Security . . . 3

2.1.3 GRAM . . . 4

2.1.4 GASS . . . 5

2.1.5 GridFTP . . . 5

2.1.6 MPICH-G2 . . . 6

2.2 Grid Job Management Systems . . . 6

2.2.1 Grid Portals: . . . 6

2.2.2 Condor–G . . . 7

2.3 Parallel Program Checkpointing . . . 8

2.3.1 CoCheck . . . 9

2.3.2 MPICH-V . . . 10

2.3.3 Egida . . . 11

2.3.4 MPI–FT . . . 11

2.3.5 FT–MPI . . . 12

2.3.6 Starfish . . . 12

2.3.7 MPI/FT . . . 12

2.3.8 Déjà vu 13

3 Motivation and Approach **15**

3.1 Need for Transparent Fault Tolerance and Migration in Grids 15

3.2 Existing Grid Job Management Systems 16

3.3 Problem Statement and Approach 17

4 Design **21**

4.1 Overall Architecture 21

4.2 Architecture of the DQ queuing System 21

4.2.1 DQ server 23

4.2.2 DQ Monitoring System 24

4.2.3 Fault Tolerance 27

4.2.4 Job Restart Mechanism 28

4.2.5 Job State Machine 29

4.3 Architecture of the Grid Job Enactor and Management Service(GEMS) 30

4.3.1 Job Submission 30

4.3.2 Job Monitoring and Migration 31

5 Implementation Issues **33**

5.1 Implementation of DQ Queuing System 33

5.1.1 DQ Server Data Structures 33

5.1.2 DQ Functionality 34

5.2 Implementation of GEMS 42

5.2.1 Job Submission 43

5.2.2 Job Monitoring 46

5.2.3 Job Migration 47

6 Conclusions and Future Work **48**

6.1 Conclusions 48

6.2 Future Work 49

LIST OF FIGURES

2.1 Checkpointing systems for parallel programs [9] 9

4.1 Overall Architecture of GEMS 22
4.2 Cluster managed by DQ 23
4.3 Threads within the DQ server 23
4.4 Job startup in DQ 25
4.5 DQ monitor architecture 26
4.6 Job restart in DQ 28
4.7 DQ job state machine 29
4.8 Job submission to a Grid resource using GEMS 31
4.9 Job migration across resource pools using GEMS 32

5.1 Example queue and node configuration files 35
5.2 nq options 36
5.3 GEMS modules 43
5.4 Sample SDL file 44
5.5 GRAM job states 46
5.6 GRAM job states with DQ 47

LIST OF TABLES

5.1 SDL tags 43

5.2 SDL attributes 45

Chapter 1

Introduction

Grid computing [12] infrastructures connect diverse set of resources such as supercomputers, clusters and data storage systems spread across multiple locations. They provide the scientific community with huge computational power to solve complex problems. Grids are highly dynamic in nature. Resources join and leave a grid without prior notification. Also the resources are heterogeneous in nature. Resources with a variety of machine architectures and operating systems are interconnected through the Grid. A set of common Grid protocols have been developed for accessing these vastly heterogeneous resources.

A Grid job management system hides the heterogeneity of the Grid and the complexity of Grid protocols from the user. It provides a usable interface to the Grid. The Grid job management system accepts job requests from a user, finds a suitable Grid resource, submits the job and transfers the output files back to the user upon job completion. The users need not bother about the location of resources, the mechanisms required to use them and keeping track of their job status. They can just submit a job to the Grid job management system that handles all these tasks on their behalf.

As the number of components in a distributed system increases, the component failure rate also increases. This is true even in the case of the Grid. The highly dynamic nature of the Grid requires novel user transparent checkpointing/restart protocols to protect the applications from machine failures. Since a huge number of resources are available on the Grid, checkpointing/restart mechanisms are also useful to migrate the applications to better resources. Significant advances have been made in defining the security infrastructure and middleware for Grids. However, fault

tolerance issues for Grid applications have been neglected.

1.1 Thesis

Application fault detection and recovery is essential to realize the true power of Grids. This thesis attempts to address the issue of application fault detection and migration across Grid resources. We describe the prototype implementation of a Grid job management system that utilizes a local resource manager to detect application failures and migrate the applications from one resource to another. The goals of this research are the following:

- A fault tolerant Grid job management system

  The Grid job management system accepts a job request from the user, searches for an appropriate resource and submits the job. As the job executes in the resource pool, one of the job processes may fail due to machine failure. The Grid job manager detects that the job has failed. If the job cannot be restarted in the current resource pool on a new set of machines, the Grid job manager migrates the job to a new resource pool where it can be restarted. As the checkpoint/restart protocols for Grid jobs evolve, the Grid job management system will utilize them to restart the migrated jobs from a previous checkpoint. Apart from these fault detection and migration services, the Grid job management system accepts a user job, submits it to an appropriate resource and transfers the output files back to the user upon job completion.

- A fault tolerant local resource manager

  The Grid job management system requires the support of the local resource manager to detect job process failures. The local resource manager has to detect the machine failures in the resource pool and suspend the execution of the failed jobs. If possible the jobs are restarted, else they remain suspended. The job state has to be reported to the Grid job manager.

The remainder of the thesis is organized as follows. Chapter 2 describes the current Grid job management systems and checkpointing approaches. Chapter 3 emphasizes our motivation and approach in designing a fault tolerant Grid job management system. Chapter 4 and Chapter 5 present the design and implementation of a prototype system. Chapter 6 concludes with future research directions.

Chapter 2

Related Work

2.1 Middleware for Executing an Application on the Grid

Grid computing provides revolutionary technologies for sharing resources across geographically dispersed organizations. Grid technologies promise to change the way organizations tackle complex computational problems. In this section we examine some of the Grid protocols implemented in the Globus toolkit for Grid computing. The Globus toolkit's API is used to build higher level services.

2.1.1 Monitoring and Directory Service

The Monitoring and Directory Service (MDS) [7] is used to find the resources that are part of the Grid. The resources notify that they are part of the Grid using the Grid Resource Registration Protocol (GRRP). Information about the resources that are part of the Grid can be obtained using the Grid Resource Information Protocol (GRIP).

2.1.2 Grid Security

The Grid Security Infrastructure (GSI) [11] is based on the public key encryption, X.509 certificates, and the Secure Sockets Layer (SSL) communication protocol. All the grid users have certificates and mutual authentication takes place using SSL protocols. The enhancements that GSI provides to the SSL protocol are delegation and single sign-on.

Usually the jobs that a user wants to run on the Grid require several resources. Some agent such as a resource broker has to request these resources on the user's behalf. For performing authentication/authorization with each of the resources the user has to be prompted for a password to decrypt his private key. To avoid this repeated password prompting, GSI creates a *proxy*. A proxy consists of a new certificate with a new public key which is signed by the owner and a new private key. Proxies have limited lifetime and thus a proxy's private key need not be stored as securely as the owner's. So the proxy private key can be stored unencrypted with proper permissions, so that an agent can access it and use it to authenticate to the resources without prompting the user for passwords each time.

2.1.3 GRAM

Globus Resource Allocation Manager (GRAM) [8] is the resource management component of the Globus toolkit. It is used for submitting jobs to remote resources. GRAM consists of the following components:

- **GRAM client:** The GRAM client is used to interact with the GRAM gatekeeper at a remote site to perform mutual authentication based on GSI and transfer a request. The requests are in the form of Resource Specification Language (RSL) [8].
- **Gatekeeper:** The gatekeeper is the component of GRAM that resides on all the grid resources and accepts requests from clients. It performs the mutual authentication of the request and resource, determines the local user name for the user and starts a job manager for the job.
- **Job Manager:** The job manager is responsible for actually creating the job processes by submitting the request to underlying local resource management systems like PBS, loadleveler, etc. The generic job manager is the fork job manager which can interact with any scheduling system by just executing the command in RSL. The job manager interacts with the local resource manager through a PERL module. The module consists of PERL scripts for submitting a job, finding status of a job and canceling a job. Depending on the request it receives, the job manager executes the corresponding PERL script. The PERL script translates the request to a command that is understood by the local queuing system, executes the command and returns the output to the job manager. To interface any new scheduler with the job manager,

a PERL module specific to the scheduler has to be written. This is called the job manager interface. Once the processes are created, the job manager monitors the processes and notifies the callback contact of any important state changes.

- **GRAM reporter:** GRAM reporter updates the MDS with information about a local resource scheduling system. Typical information includes total number of nodes, number of nodes currently available, currently active jobs, and expected wait time in a queue.

2.1.4 GASS

The Global Access to Secondary Storage (GASS) service [4] provides mechanisms for transferring data between remote HTTP, FTP, or GASS servers. GASS is used to stage executables and input files before job startup and to retrieve output files when the job is done. It uses GSI for authentication.

2.1.5 GridFTP

Though GASS can transfer data between a user's machine and a Grid resource, it is not efficient for transferring huge amounts of data. So the most popular internet file transfer protocol (FTP) has been enhanced to better suit the requirements of Grids. The following are the features of GridFTP [2]:

- GSI security on control and data channels
- Multiple data channels for parallel transfers
- Partial file transfers
- Third-party (direct server-to-server) transfers
- Authenticated data channels
- Reusable data channels
- Command pipelining

2.1.6 MPICH-G2

MPICH-G2 [17] is a MPI library for creating parallel programs for Grid environments. It extends the existing MPICH implementation of MPI with Globus toolkit services. The Globus services are used for authentication/authorization and job submission and monitoring to resources across multiple organizations. This helps to hide the heterogeneity of Grid environments from the user.

2.2 Grid Job Management Systems

The Globus toolkit provides various command line tools for basic job submission and monitoring. The jobs submitted to Grid resources typically run for many hours or even days. It is a cumbersome task for a user to manually log into various resources and submit/monitor his jobs over such a long period. Grid job management systems solve this problem by providing a usable interface to access Grid resources. The user submits his job request to a Grid job management system. It searches for suitable resources, submits the job, monitors it on behalf of the user and finally transfers the output to the user's machine. The user can log into the Grid job management system and check the status of all his jobs running on various resources. He need not log into each resource individually. Thus the Grid job Management systems hide the heterogeneity of the Grid from the user. They can be considered as a batch system to the Grid resources. This section examines the existing Grid job Management systems.

2.2.1 Grid Portals:

Grid portals provide a convenient web interface into which users can login and submit their job requests.

Grid Portal Development Kit (GPDK) [21] facilitates the rapid development of such portals. GPDK provides a set of Grid service beans implemented using the Java Beans technology and the Java CoG kit [18] to access all the grid services. In addition, portals developed with GPDK also utilize the usual web portal technologies like Java Server Pages(JSP), Tomcat and Java Servlets. They can be viewed as a web interface to the Globus commands.

GridPort [25] also facilitates the development of Grid portals. GridPort uses the Perl CoG kit to access Grid resources. The web pages and data are built from server-side Perl modules or libraries, and simple HTML/JavaScript on the client side, so they can be be easily viewed from any browser.

Grid portals do not support persistent job queues. Also they do not have interfaces to support job migration.

2.2.2 Condor–G

Condor–G [13] provides a computation management agent that resides on the user's machine. The agent is used to submit a job request to the Condor-G scheduler. The scheduler finds the suitable resource by searching a predefined list of resources. If a suitable resource is not found the job waits in a persistent job queue and is scheduled later when the resource becomes available. When a resource is available, the scheduler creates a GridManager, which interacts with the Gatekeeper of the resource to perform authentication. Then it stages in the job files using GASS. The Gatekeeper then creates a job manager which submits the job to the local job scheduler. The GridManager periodically queries the job manager to find the state of the job. The GridManager notifies the important changes like failures or job completion to the user through email. The GridManager also manages the proxy credentials of the user or it can be configured to use the MyProxy server. The GridManager as well as the job manager cease to exist after the execution of a job is completed.

Condor–G handles four kinds of failures: crash of the Globus job manager, crash of the machine that manages the remote resource (the machine that hosts the gatekeeper and job manager), crash of the machine on which the GridManager is executing (or crash of the the GridManager alone), and failures in the network connecting the two machines. The GridManager periodically probes the job manager of its job to find the status of the job. If the job manager does not respond, it tries to contact the gatekeeper of the resource. If the gatekeeper also does not respond, then it is assumed that the machine that manages the remote resource has crashed. GridManager waits until the machine comes up and then requests the gatekeeper to start a new job manager. The gatekeeper is automatically started when the machine comes up as it is usually configured as a system start up process. If the gatekeeper responds, then only the job manager has failed and the GridManager requests the gatekeeper to restart the job manager. If there is a network failure the GridManager waits until it can connect to the job manager as already described. However, the job manager might have exited if the job has finished execution. The GridManager is unaware of this and thinks that the job manager has failed, so it requests the gatekeeper to restart the job manager. The job manager, when restarted notifies the GridManager of job completion. Local failures of the GridManager itself are handled by checkpointing the GridManager process by using Condor stand alone single process

checkpointing library.

The fault detection mechanisms make the Condor–G system fault tolerant but they do not help in detection of application failures. Suppose a parallel job is submitted to a resource but one of the job processes is killed because the machine on which it is running crashes. The other processes of the job that are alive keep waiting for some messages from the failed job and the application hangs. If the resource management system cannot detect the failure, the job is killed only after it exceeds its time limit or by human intervention. Condor–G cannot detect that the job has failed inside the resource. Also Condor–G has no interfaces to support job migration across resources.

2.3 Parallel Program Checkpointing

There are a number of efforts to provide fault tolerance for parallel programs based on MPI. These can be broadly classified into the following two groups:

- Checkpointing Protocols: In these protocols each process periodically saves its state to a stable storage. The processes must be coordinated in some way while taking a checkpoint, to provide consistent global checkpoints. If this is not taken care, then the checkpoints may be not consistent due to the messages in transit between the processes.

- Log based Protocols: Logging protocols are of three types - pessimistic, optimistic and causal. In pessimistic protocols the processes are allowed to communicate only from recoverable states. Any critical information is logged before allowing the processes to communicate. In optimistic protocols, processes are allowed to communicate with other processes even from states that are not recoverable. These protocols take care that these states will eventually become recoverable, if no faults occur. Causal protocols allow restarts from unrecoverable states also, provided the correct processes do not depend on that state.

A detailed survey of the various methods for checkpointing message passing systems can be found in [14]. We examine some of these systems in the following sections. Figure 2.3 categorizes these methods in terms of their basic approach and the level at which they operate.

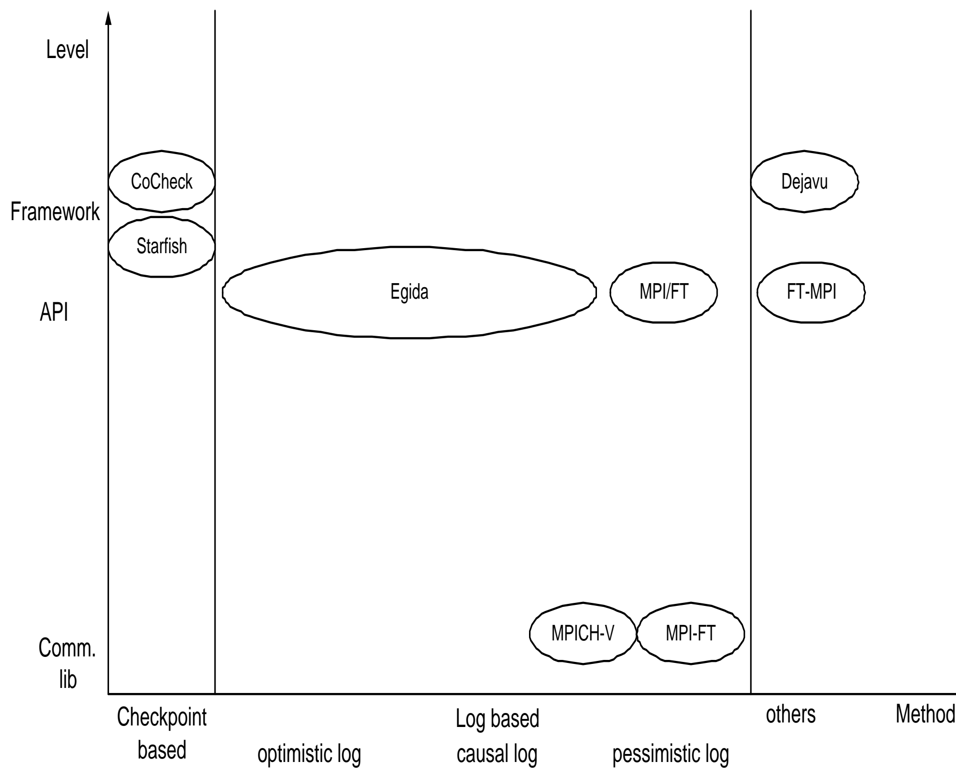

Figure 2.1: Checkpointing systems for parallel programs [9]

2.3.1 CoCheck

CoCheck [24] is used in Condor [19] for checkpointing and migrating MPI applications. CoCheck uses the Condor single process checkpointing mechanism for checkpointing the state of the individual processes. To handle the messages that are in transit when taking a checkpoint, CoCheck uses special messages called "ready messages" (RM). Whenever a checkpoint is to be taken or a set of processes have to be migrated, each process gets a notification. Each process then sends a RM to all the other processes. Besides, each process receives any incoming message. If it is a RM from a certain process, then it can be concluded that there will be no further messages from that process. If the message is not a RM, it is stored in a reserved buffer area of the process. When finally a process has collected the RMs from all other processes, it can be sure that there are no further messages on the network which have its address as destination. All messages that were in transit are safely stored in the buffer and are available for later access.

Each application process has to maintain an address mapping table. This is required because, when the application process is migrated it will be restarted on a different machine with a new process id. CoCheck assigns a virtual address to each application process initially. Whenever one application process communicates with another, the CoCheck library maps the virtual address to the actual address of the process by looking up the process's mapping table. To hide all these additional functionalities from the user and provide transparent checkpointing, CoCheck provides wrappers to all the functions in MPI library. This implementation of MPI is called tuMPI.

2.3.2 MPICH-V

MPICH-V [9] consists of a communication library based on MPICH [15] and a runtime environment. The MPICH-V library can be linked with MPI programs and it reimplements all the subroutines provided by MPICH to provide the checkpointing capabilities. The runtime environment consists of a dispatcher, channel memories, checkpoint servers and computing nodes.

MPICH-V relies on the concept of Channel Memory (CM) to provide fault tolerance. CMs are dedicated nodes which handle the communication between nodes. Each node has a designated CM, called home CM, from which it receives messages. If a node has to send a message to another node, it sends the message to the receiver's home CM. The CM then saves a copy of the message before sending it to the receiver.

The Condor Stand-alone Checkpointing Library (CSCL) is used to checkpoint the state of the processes on the nodes periodically and these checkpoint files are stored on checkpoint servers. When a failed process is restarted from a checkpoint image, the messages that the process has sent out after the checkpoint has been taken have to be replayed. So with every checkpoint, information about the message from which the replay should start is stored. All these messages can be found in the node's CM, since all communication messages are stored in the CM before sending to the receivers. Hence for every process, an execution context is saved in its checkpoint server and a message context is saved in home CM.

When the system initializes, MPICH-V runtime provides each node with the information about its home CM and checkpoint server. The nodes send periodic "alive" messages to the dispatcher. The dispatcher can detect node failures through missing "alive" messages. The dispatcher restarts the failed processes from a previous checkpoint on a new node. It contacts the failed node's checkpoint server to obtain the checkpoint image. The dispatcher provides the restarted process with

information about the failed node's home CM so that messages for replay can be obtained. Thus the failed processes can be successfully restarted.

The major drawback of this system is that it assumes all the CMs are free from failures. Since the CMs are key components involved in handling messages, the entire system fails if a CM fails.

2.3.3 Egida

Egida [23] is an object oriented toolkit supporting transparent log-based rollback-recovery. MPI programs have to be relinked with the Egida library for taking advantage of its fault tolerant capabilities. No other source code modifications are necessary. Egida treats all the communication, fault detection and checkpoint/restart operations as events and provides appropriate handlers for all these operations. All the MPICH library calls are reimplemented for this purpose. Egida has been ported to MPICH by replacing the P4 communication layer of MPICH with Egida's object modules. The P4 layer implements the send/receive operations in MPICH. Egida checkpoints the process state as well as all the messages which leads to large overheads.

2.3.4 MPI–FT

MPI–FT [20] is built from LAM–MPI [6]. Process failures are detected using a monitoring process called observer. There are two modes in which MPI–FT handles checkpointing of inflight messages:

- Distributed Buffering: Each process saves the messages it sends to others. Each process maintains a 1-dimensional matrix where the number of messages received from peer processes is stored. When a new process is started as a replacement to a failed process, all the living peers have to resend some of the messages to the new process. Every living peer process consults its matrix and decides the messages it has to resend based on the counters in the matrix.

- Centralized Buffering: In this mode all the messages are stored in the observer. While restarting a failed process, the observer decides what messages to resend to it.

Significant modifications of the MPI communicator and collective operations were required to implement these message checkpointing strategies.

2.3.5 FT–MPI

According to MPI semantics the failure of a MPI process or communication causes all communicators associated with them to become invalid. FT–MPI [10] extends the MPI communicator states to accommodate some error states. When a failure occurs, a communicator goes to an invalid state and any further MPI calls attempted in the application are returned one of the error states. It is up to the application to handle the failure. FT–MPI does not perform any kind of application checkpoint/recovery.

2.3.6 Starfish

Starfish [1] provides a parallel execution environment that adapts to dynamic changes in a cluster environment. In a Starfish runtime environment each node in the cluster runs a Starfish daemon. These daemons can communicate with each other using the Ensemble group communication toolkit [5]. These daemons are responsible for MPI job startup and monitoring the application processes. Starfish uses an event model where processes and components register to listen on events. The event bus provides messages reflecting changes in cluster configuration (addition and deletion of nodes) and process failures. For example, when a node is added the daemon on that node generates a *view* event which is notified to all the other daemons.

For checkpoint/recovery Starfish provides users with two options - a coordinated system level approach or an uncoordinated application level approach. The system level approach is based on the Chandy-Lamport algorithm and can be used only for some trivial parallel applications. Issues like inflight messages and consistency of communicators are not addressed by this approach. In the application level approach application recovery is left to the user. Hence Starfish provides only a powerful fault detection mechanism but it lacks efficient checkpoint/restart mechanisms.

2.3.7 MPI/FT

MPI/FT [3] provides fault tolerance based on the model of the parallel program. MPI/FT provides checkpoint/recovery strategies only for master-slave applications with a star topology and SPMD applications with an all-to-all topology.

A common strategy for application fault detection is followed for both application models. An MPI/FT job process consists of two threads. One is the application thread and the other is called

a self checking thread(SCT). SCT is responsible for detecting failures. In addition there is a coordinator thread in the application process with rank zero. The SCTs from all processes send periodic heartbeat messages to the coordinator. The coordinator detects process failures through missing heartbeats. Certain machines in the resource pool are reserved for restarting failed processes. The coordinator obtains a new machine to restart a failed process from this set of reserved machines.

For master-slave applications with star topology, checkpointing an application process is trivial as there is no communication between the slaves. MPI/FT leaves checkpointing of this kind of applications to the application itself.

For SPMD applications with all-to-all topology, MPI/FT provides a synchronous collective operation for producing globally consistent checkpoints. Consistency of checkpoints is established by requiring the participation of all processes in the collective operation and the checkpoint operation returns only when every process has successfully taken a checkpoint. A complementary collective operation for reading data from a previously saved checkpoint is also provided.

Thus MPI/FT provides automatic fault tolerance and recovery to a small subset of parallel applications.

2.3.8 Déjà vu

Déjà vu provides fault tolerance to any kind of message passing library. It uses the Weaves compiler framework [26] for providing checkpoints portable over homogeneous environments. Weaves post compiler analysis is used to capture the global state, registers and function invocation history of a process. To capture the dynamic memory allocated, a library of overloaded memory allocation system calls is used. For handling the messages in transit, a user level implementation of the TCP/IP stack is used. Since TCP is a reliable transmission protocol, it will correctly handle all the error conditions arising due to communication between a pair of nodes. The user level stack also provides a virtual network to which the processes bind. Whenever the nodes communicate, they use these virtual addresses which remain the same through out the application execution. The virtual address is then translated to the actual addresses which changes when a process is migrated using address translation tables. So the application execution can proceed independent of the physical platform. Déjà vu also employs a novel incremental checkpointing algorithm to reduce the size of checkpoint files.

All the systems described above except for Déjà vu, modify some implementation of MPI (MPICH

or LAM-MPI) or provide an API which can be used to modify the applications to provide fault tolerance.

Chapter 3

Motivation and Approach

3.1 Need for Transparent Fault Tolerance and Migration in Grids

Fault tolerance is an important aspect of any large scale parallel application even at the level of a single cluster. In most parallel applications, if a machine on which one of the job processes is running goes down, the rest of the job processes keep waiting for the messages from the failed process and the entire application hangs. In the absence of any fault detection and recovery mechanisms, the resource management system kills the job at the end of its time limit, thus wasting the allocated resources. In the case of Grids, not only protecting the application from machine failures is important; it may be desirable to migrate an application from one resource to another that better suites the application requirements. The application has to be restarted from a previous checkpoint on the new resource. Since Grids include resources spread across multiple organizations, checkpointing and restart may be needed to enforce scheduling priorities. For example, the local resource manager of a resource belonging to a particular organization might want to give more priority for jobs belonging to its own organization. If enough resources are not available for these jobs, the resource manager has to suspend the jobs submitted from other organizations to free more resources. The suspended jobs from other organizations will be later restarted from a checkpoint. Since the target applications are mostly complex scientific codes, a truly transparent (system level) checkpointing approach that does not require modification to application codes is required. The Déjà vu system described in the

previous chapter aims at providing such a user transparent checkpointing approach for large scale parallel applications. A Grid job management system can utilize such a checkpointing approach and provide job migration across the Grid resources.

3.2 Existing Grid Job Management Systems

In a Grid environment user jobs are scheduled on multiple resources spread across different organizations. It is very inconvenient for the user to log into the various resources, submit jobs, and later periodically check their status. A Grid job management system does these functions on behalf of the user. It hides the heterogeneity and the complexity of Grid protocols from the user. The user logs into the system and submits a job. The Grid job manager in turn finds a suitable resource for the user's job, submits it and then periodically monitors the status of the job. The user can log into the Grid job management system and check the status of his jobs just like he does when he submits a job to any single resource. The Grid job management system encapsulates the heterogeneity of the Grid and provides the user with a single resource view of the Grid. Condor-G and portals built using GPDK are the most popular of existing Grid job management systems as explained in the previous chapter. Here we examine some of the shortcomings of these systems.

Portals developed using GPDK just provide a convenient web interface for all Globus commands. They do not have a persistent queue of jobs and have limited functionality in terms of resource discovery. They just query the MDS and give the user a list of the available resources. Existing Grid portals cannot be configured with a resource broker for more sophisticated resource discovery. Also the portals cannot detect any kind of job failures.

Condor-G is a robust Grid job manager that can submit jobs to various resources and has some fault tolerant capabilities, as described in the previous chapter. However, these capabilities do not aid in detection of application failures. The Condor-G manual lists the following limitations:

- No checkpoints. Condor-G cannot detect job process failures and restart them on a different machine or resource from a previous checkpoint. The future releases of Condor–G plan to address this using Condor's single process checkpoint mechanism. But for parallel programs the checkpointing strategy that Condor would provide uses the CoCheck protocol described in the previous chapter. In that case applications designed using MPICH, which is the most popular MPI library, cannot be used as CoCheck requires the applications to use a special

MPI library called tuMPI. This may not be acceptable to many users and also tuMPI is not actively under development. Thus condor–G has limited ability to checkpoint MPI programs due to lack of robust checkpointing mechanisms.

- No matchmaking. Condor–G does not use the Condor system's matchmaking capabilities for resource discovery based on application requirements. It just uses MDS to obtain a list of resources from which the user has to choose the resource on which the job can be run.

- File transfer is limited. There are no file transfer mechanisms for files other than the executable, stdin, stdout, and stderr. This is because Condor–G does not use GridFTP protocol for transferring files. It uses GASS, which is the Globus toolkit component for transferring files of small sizes. GASS is not optimized for transferring large files, which is why GridFTP has been developed.

- No job exit codes. The users cannot know if their job failed due to a bug in the application code or due to a machine failure. This is because Condor–G does not employ any application failure detection mechanisms.

In essence, both Condor-G and the GPDK based web portals can submit the user jobs to Grids and monitor them to find if they are running or done. They cannot detect job process failures and hence cannot help in migrating the application to a better resource.

3.3 Problem Statement and Approach

The goal of this thesis is to develop a Grid job management system that can efficiently detect application failures and migrate them to a new resource pool where they can be restarted, in case the application cannot be restarted in the current resource pool. Apart from these fault detection and migration capabilities, the Grid job management system provides the usual services like job submission, resource discovery and monitoring the application for completion. As novel checkpointing algorithms such as Déjà vu become available, the Grid job management system would utilize them to restart the jobs from a checkpoint. Hence the design of the Grid job management system should facilitate easy plugging in of checkpoint/restart modules. The thesis also aims at defining the requirements that a local resource management system has to satisfy to participate in fault de-

tection and migration of Grid applications. A Globus job manager interface for such a local resource manager also has to be developed.

Once the Grid job management system submits a job to the local resource manager, it has no control over the job processes. The Grid job management system contacts the Globus job manager to find the status of the job periodically. The job manager in turn queries the local resource manager and returns the status of job to the Grid job management system. Thus it is the local resource management system that has to detect job process failures and push the information to the Grid job management system. The features of the local resource manager needed for efficient fault detection and job restart are described below.

Any job submitted to a typical local resource manager goes through the following states:

- wait: The job is yet to be scheduled by the local resource manager.
- running: The job has been allocated resources and started execution.
- done: The job has finished execution successfully.
- failed: The job has encountered some fault.

When a machine running a single process job goes down, the job status immediately goes to the "failed" state. When one of the job processes of a parallel application is killed the job does not go to the "failed" state immediately. This is because parallel jobs have multiple processes. The other job processes are still alive, waiting for messages from the killed process. The application hangs and the local resource manager kills the job after the time limit for the job exceeds. This leads to wastage of the resources allocated. A fault-tolerant local resource manager has to be able to detect this situation for parallel jobs. For this we introduce a new state called "restart". When one of the job processes of a parallel job dies, a fault tolerant local resource manager should suspend the job's execution by killing the rest of the job processes. The job goes to the "restart" state. The local resource manager sets a certain number of machines in the resource pool aside for restarting failed processes. This is called a "restart pool". A failed parallel job retains the resources originally allocated to it. The machine to restart the failed job process is obtained from the restart pool and the job is restarted from a previous checkpoint. If no machines are available in the restart pool, the job waits in the "restart queue" along with other failed jobs. As long as the job is in the restart queue its state is "restart". Once it is restarted, it goes to a "running" state again.

The Globus job manager uses a PERL module, that contains PERL scripts to interact with the local resource manager. This PERL module is called the job manager interface to the local resource manager and it contains separate PERL scripts for submitting a job, canceling a job, finding the status of a job. The job manager executes a specific PERL script based on the request it receives. The output is returned to the job manager which in turn returns the output to the contact. A new Globus job manager interface that can interact with this kind of local resource manager and report the job states correctly to the job manager is also required. The job manager in turn reports the state of the job to the Grid job management system.

In summary the following are the features of a local resource manager that can detect application failures and report them to the Grid job manager:

- A monitoring system in the resource manager detects machine failures.
- The jobs whose processes were running on the failed machine are suspended by killing all the other job processes and the job goes to a "restart" state.
- The job is restarted by obtaining a new machine from a restart pool to replace the failed machine. If no machines are available in the restart pool the job remains in a "restart" state.
- A job manager interface reports these job state changes to the job manager that in turn reports it to the Grid job manager.

Given such a local resource manager that can detect job failures, a Grid job management system with the following features can be implemented. We call our prototype implementation Grid Enactor and Management Service(GEMS).

- Job monitoring and fault detection: GEMS periodically contacts the job manager to obtain the status of the job. If the job fails, the local resource manager places it in the restart queue. The job manager then reports the state as "restart". Hence GEMS comes to know that the job has failed.
- Job migration: If the load on a particular resource pool is high, it is likely that all the machines in restart pool are used up. So a failed job remains in the restart queue for a long time. GEMS can detect this as it keeps monitoring the job and sees that it has been in "restart" state. If a resource where the job can be restarted immediately is available, GEMS can initiate migration of the job to that resource.

- Job submission on behalf of the user: GEMS does this using the GSI protocols.
- File transfer: GEMS utilizes GridFTP to transfer files input/output files of large size between the resource and the user's machine.
- Resource discovery based on user requirements: GEMS utilizes the services of a resource broker which searches for resources based on the user's requirements, rather than just giving a list of resource options to the user from which he has to choose.

Thus GEMS overcomes the inadequacies of existing Grid job management systems.

Chapter 4

Design

4.1 Overall Architecture

Figure 4.1 shows the overall architecture of the GEMS system.

The user submits a job request file written using Simulation Definition Language (SDL) [16] to the Grid Job Enactor and Management System (GEMS). Based on the job requirements, GEMS finds the appropriate resource pool with the help of a resource broker or meta scheduler. We assume that the local queuing system in each resource pool is DQ. Once the job is submitted to a resource pool, DQ allocates the required machines and starts the job. If any of the machines on which the job processes are running fails, DQ suspends the job execution and tries to restart it on a new set of machines based on the availability of machines in the resource pool. We call this suspended state the "restart state". GEMS periodically monitors the status of the job. If it detects that the job has been in the restart state for a long period, it tries to find a new resource pool to which the job can be migrated.

4.2 Architecture of the DQ queuing System

DQ is a queuing system for the management of clusters or networks of workstations. Jobs submitted to DQ must be specified as belonging to a specific queue depending on their type. System administrators can define the queues in a configuration file called *queue.conf*. The machines in the resource

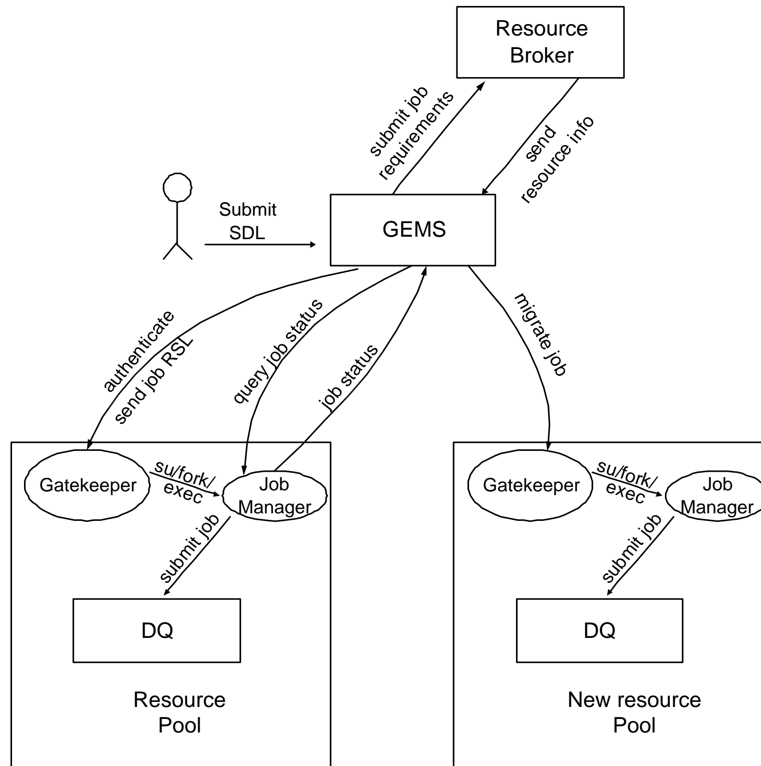

Figure 4.1: Overall Architecture of GEMS

pool support one or more of these queues as defined in another configuration file called *node.conf*. An example set of queues is

- Production queue: For long running jobs.
- Experimental queue: For short lived jobs.
- Restart queue: For jobs that are in the restart state.

DQ has two main components: the DQ server and the monitor clients. Figure 4.2 shows how DQ runs on a cluster. The DQ server resides on the head node of the cluster and all the other nodes of the cluster run a monitor client. The detailed description of these components is given in the following sections.

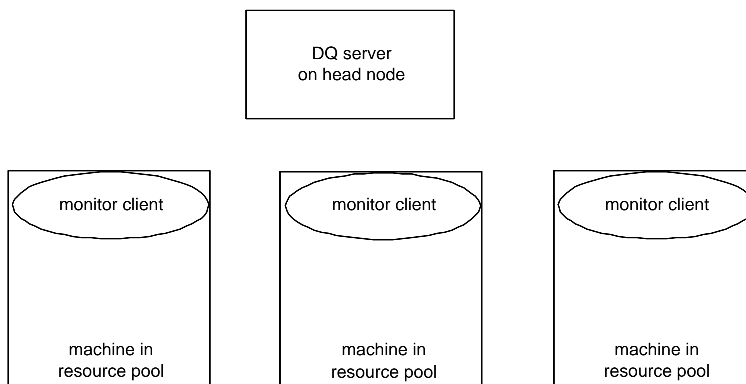

Figure 4.2: Cluster managed by DQ

4.2.1 DQ server

The DQ server is multi-threaded as shown in Figure 4.3. The following paragraphs explain the

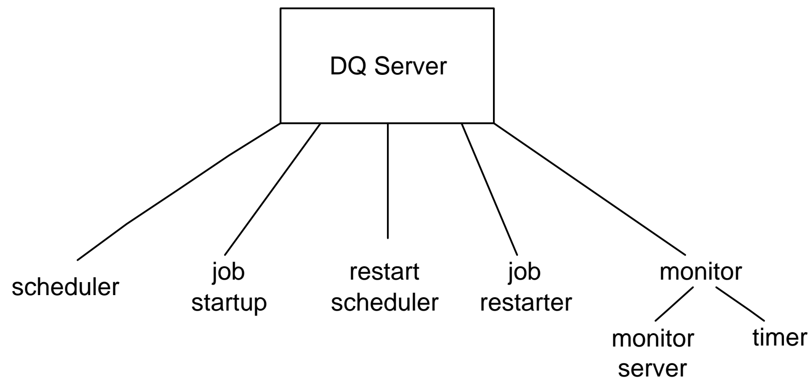

Figure 4.3: Threads within the DQ server

function of each thread in detail.

User Interaction

The **Server Thread** is the main thread of the system. When the system starts up this thread reads in the configuration files and starts the remaining threads. Apart from initial startup, its main function is to accept requests from the users. The users connect to the server using the client program called “nq”. The server parses the request and passes it to the appropriate thread. If a job is submitted it returns a job id to the user and signals the scheduler thread. The other functions

users can perform using nq are querying the status of the job , cancellation of their job, and viewing the job queues.

Job Scheduling

The **Scheduler Thread** assigns the machines in the resource pool for a job. The scheduling algorithm in the current implementation is very simple. The machines are allocated to the jobs on a "first come first served" basis, and the job retains the machines till it finishes execution. The scheduler finds which queue the job belongs to. If there are enough free machines that support this queue to satisfy the job requirement, they are assigned to the job. Else the job waits in the queue until more machines are available. More sophisticated scheduling algorithms that take job priorities and their running time into consideration will be implemented in the future to utilize the resources effectively.

Job Startup

The **Job Startup Thread** sends the job request to the monitor clients on the assigned machines which actually start up the job, as shown in Figure 4.4. If a MPI job is submitted, the job startup thread sets the necessary environment variables based on the underlying MPI channel and then sends the job request to the monitor clients. The monitor clients start the job executable with the parameters and environment variables sent.
The remaining threads of the DQ server are described in the following sections.

4.2.2 DQ Monitoring System

The DQ monitoring system consists of the monitoring server and the monitor clients which reside on each and every machine as shown in Figure 4.5.

Monitor Server

The monitor server is implemented as one of the threads of the DQ server. So it resides on the front end node of the cluster along with the rest of the DQ server threads. The monitor server receives messages from the monitor clients. These messages can be of two types.

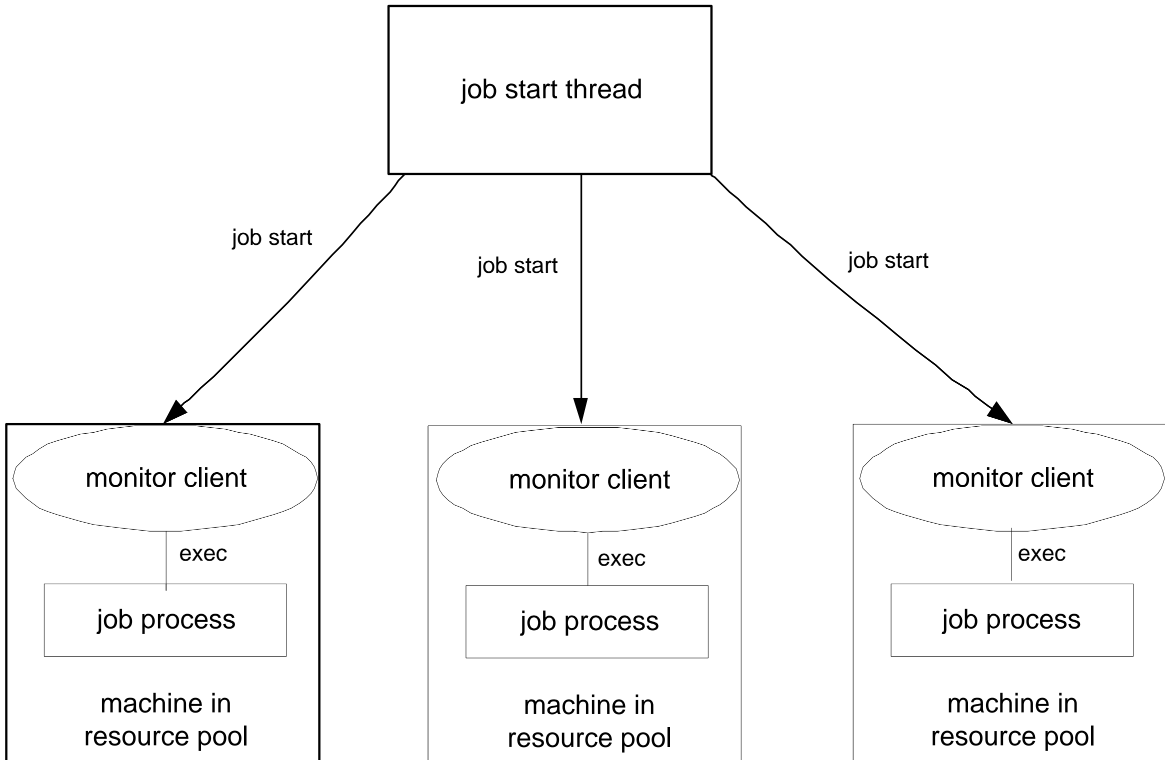

Figure 4.4: Job startup in DQ

- Job Status: The monitor client sends this message if a job process executing on its machine exits. The message specifies if the process exited gracefully or due to some fault.
- Machine Status: The monitor clients periodically send a heart beat message to the monitor server. Through this message the clients report the health of the machine on which they reside to the server. The message contains details like the number of processes executing, free memory, and the load average of the machine.

When the server receives a job status message, it checks if the job process has exited gracefully. If all the other processes of the job have finished execution gracefully, then all the machines allocated to the job are released and appropriate log files are written. If the job process failed due to some error, the job execution is terminated and the reason is written to log files.

The machine status messages are stored by the server for the system administrators who can use the information for monitoring the resource pool. The timer thread periodically checks to see if

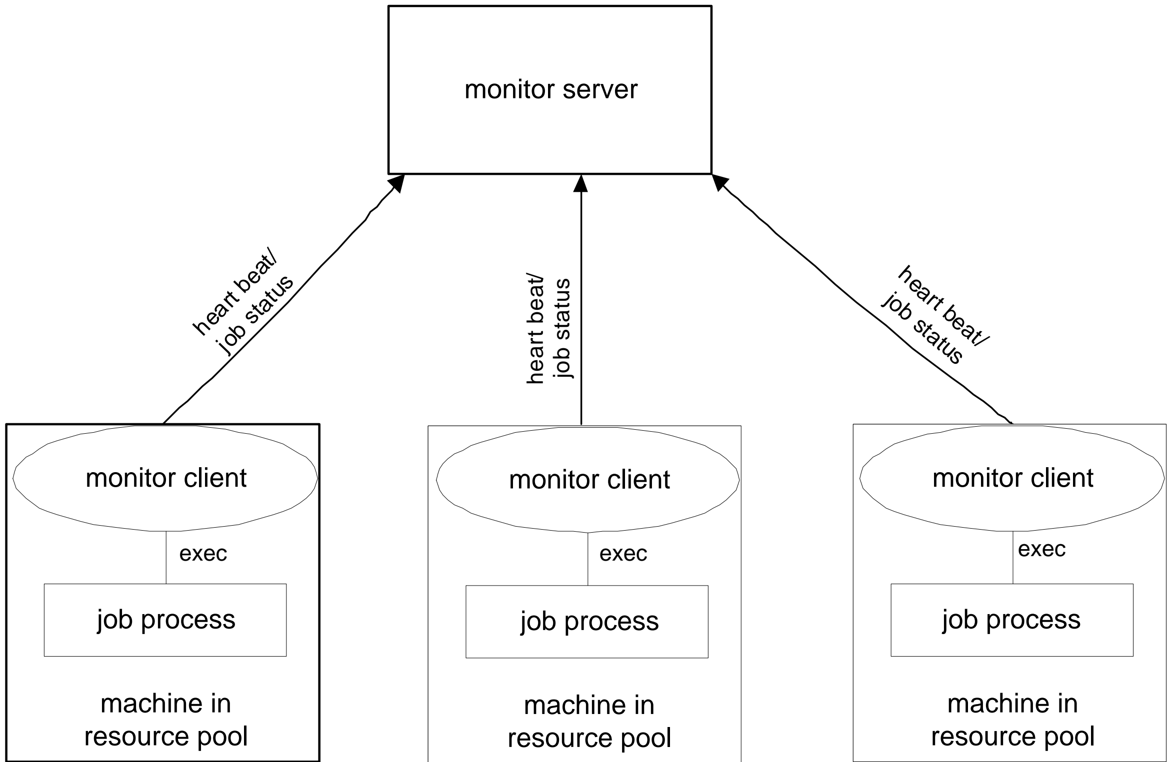

Figure 4.5: DQ monitor architecture

messages have been received from all the machines. If messages are not received from a machine for a long period then the timer thread confirms that the machine went down. It marks the machine as offline and initiates the restart process for the jobs running on the machine.

Monitor Clients

The monitor clients perform two important functions:

- Job Startup: The monitor client receives the job startup request from the job startup thread in the DQ server. The client then starts up the job in a separate process using the exec system call.
- Health Monitoring: The monitor client collects information about the machine on which it runs and periodically sends it to the monitor server. These heart beats are UDP messages and

the time interval between the heart beats can be configured. The heart beats do not result in much traffic within the resource pool.

4.2.3 Fault Tolerance

A MPI job can fail if any of the job processes fail. The rest of the job processes keep waiting for the messages from this process and the application hangs. The DQ monitoring system handles two modes of failure for MPI jobs. This section describes these failures and how the monitoring system detects them.

- Hardware Failures: If a machine on which the process is running goes down then the job process is killed. The job process cannot communicate with its peers if any hardware failure (e.g., network card failure) occurs on the machine. In these cases the monitor client also cannot send its heartbeat message to the server. The timer thread of the monitor in DQ periodically checks if a heartbeat is received from every machine in the resource pool. If the heartbeats from a machine are continuously missing, then the monitor server concludes that a hardware failure occurred on that machine.

- Software Failure: A job process can fail due to a program error (e.g., a divide by zero error). As explained earlier the job processes are always started by the monitor clients on the machines through exec system calls. So the monitor client becomes the parent of the job process. Whenever a child exits the parent receives a signal. The parent can analyze this signal and know the reason for the child's exit. Hence the monitor client can detect if the child has exited due to an error in the application code or finished its execution gracefully. This kind of job control is not possible through the mechanisms which use rsh or ssh for job submission. In such mechanisms the job startup program on the head node uses rsh or ssh to send the job startup message to every machine allocated to the job. Thus job startup program is not the parent of any of the job processes as they are on different machines; hence it has no knowledge of the job process execution. In the job submission mechanism used in DQ, each job process has a local parent which is the monitor client. So a better monitoring of the job processes is possible.

4.2.4 Job Restart Mechanism

The job restart mechanism is shown in Figure 4.6. If one of the machines assigned to a MPI job goes

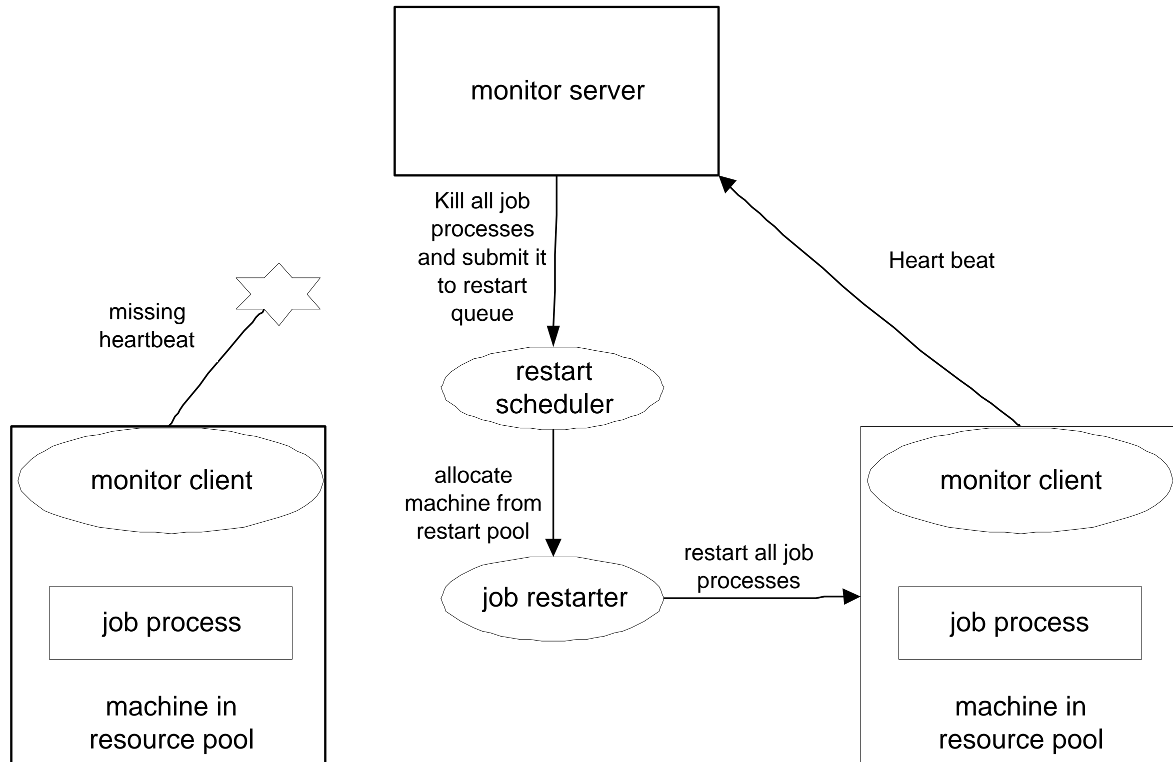

Figure 4.6: Job restart in DQ

down or a hardware failure occurs on it, then the job cannot continue its execution because all the other processes will be waiting for its messages. This is true in case of software failure also. DQ tries to restart the failed job process on a different machine if the reason for failure is a hardware failure. It is assumed that software failures may be due to logical errors in the application code which would be repeated even after a restart. So in the case of a software failure, DQ just terminates the job and writes the reason of failure to a log file associated with the job.

As explained above, the hardware failures are detected through the missing heartbeat messages. Once the timer thread detects a missing heartbeat, it identifies the jobs whose processes were executing on that machine. It then suspends the execution of these jobs by sending a job cancel message to the monitor clients on all the other machines on which the processes of these jobs are

running. The monitor clients kill the job processes. All these jobs retain the machines originally allocated to them. These jobs are moved to the restart queue and they wait in this queue until they are restarted.

The machines required for job restart are obtained from the set of machines that support the restart queue. The system administrators can decide how many machines they want to dedicate to this queue based on the machine failure rate. The restart scheduler, which is a thread in the DQ server, allocates machines to the jobs that are waiting in this queue. As soon as the restart scheduler allocates a machine from the restart pool to a job, the job restart thread sends a request to the monitor clients on all machines to restart the job processes. In the current implementation of DQ, jobs are restarted from scratch. After integration with Déjà vu, jobs will be restarted from a checkpoint.

4.2.5 Job State Machine

Figure 4.7 shows the different states through which the job goes in the DQ queuing system. The

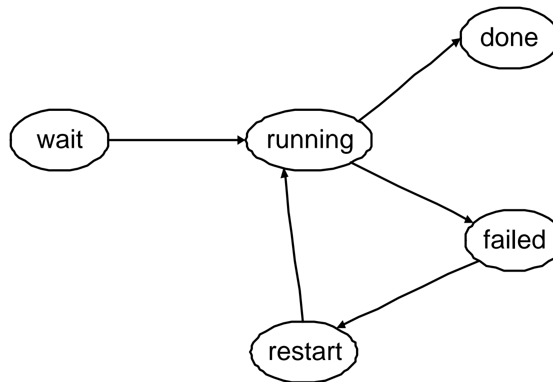

Figure 4.7: DQ job state machine

job is initially in the “wait” state in its queue. Once the scheduler thread finds enough machines that can satisfy the request it assigns these machines to the job. The job start thread starts the job by sending a message to all monitor clients on these machines. The job then goes to a “running” state. The job can go to “failed” state if a hardware or software failure occurs or it can successfully complete its execution and goes to “done” state. If the reason for failure is a hardware failure the job goes to a “restart” state. It remains in this state until the restart scheduler assigns a machine

from the restart pool. The job then goes to “running” state as soon as the job restart thread starts it again.

4.3 Architecture of the Grid Job Enactor and Management Service(GEMS)

The Grid Job Enactor and and Management Service (GEMS) is used for the submission, monitoring and management of jobs to the Grid resources. GEMS consists of a sever program that runs on a designated machine. A GEMS client program on the user’s machine is used to connect to this server and perform various tasks.

4.3.1 Job Submission

The process of job submission in GEMS is shown in the Figure 4.8. The user describes his job requirements in a file using Simulation Definition Language (SDL) [16]. More details about SDL are given in next chapter. The user must upload his proxy credentials to a MyProxy server [22] before job submission. The GEMS client program, when started asks the user for the location of the SDL file and the user name and password which the user used to store his proxy credentials on the MyProxy server.The client connects to the GEMS server and transfers this information. The server parses the information in the SDL job request and stores it in its data structures. It then contacts a resource broker with the job description. The resource broker returns the contact information of the suitable resource. The job submission module of GEMS creates an RSL for the job, contacts the MyProxy server with the user’s id and password and retrieves his proxy credentials. It then contacts the gatekeeper of the selected resource and presents the proxy credentials for authentication. The gatekeeper then creates a job manager for this job which actually submits the job to the underlying DQ scheduler that manages the resource pool. The required input files and executables of the job are then transferred by the Grid FTP module of GEMS from the user’s machine to the resource. The job starts execution whenever DQ schedules it.

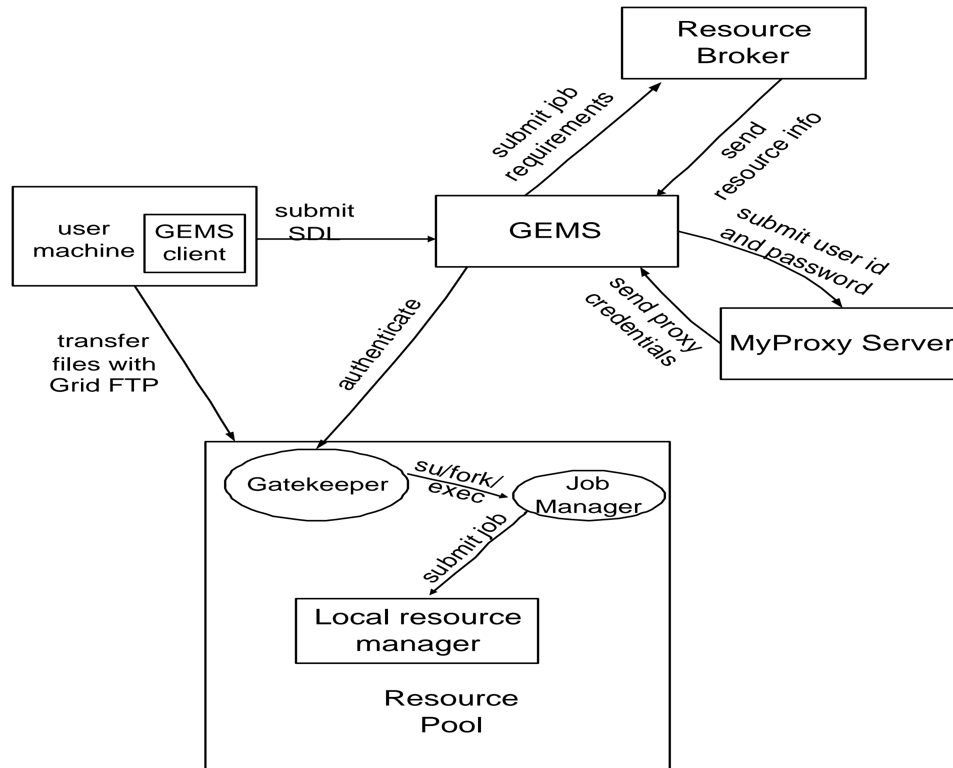

Figure 4.8: Job submission to a Grid resource using GEMS

4.3.2 Job Monitoring and Migration

Once the job is submitted, GEMS periodically monitors its status by contacting the job manager. The job manager reports the status of the job as shown in the Figure 4.9. Suppose one of the job processes fails due to the failure of a machine in the resource pool. DQ then suspends the job execution and puts the job in the restart queue. Thus the job goes to "restart" state as shown in Figure 4.7. If the load on the resource pool is high, it is possible that all the machines allocated to the restart queue are used up. In this case the job remains in restart state for a long period of time. GEMS notices this and tries to find a new resource by contacting the resource broker again. If a new resource where the job can be started immediately is available, then GEMS sends a job cancel message to the job manager. The job manager in turn notifies the DQ queuing system of the job cancel request. DQ then removes the job from the queue and all its resources are released. The GEMS job submission module starts the job submission process by contacting the new resource's

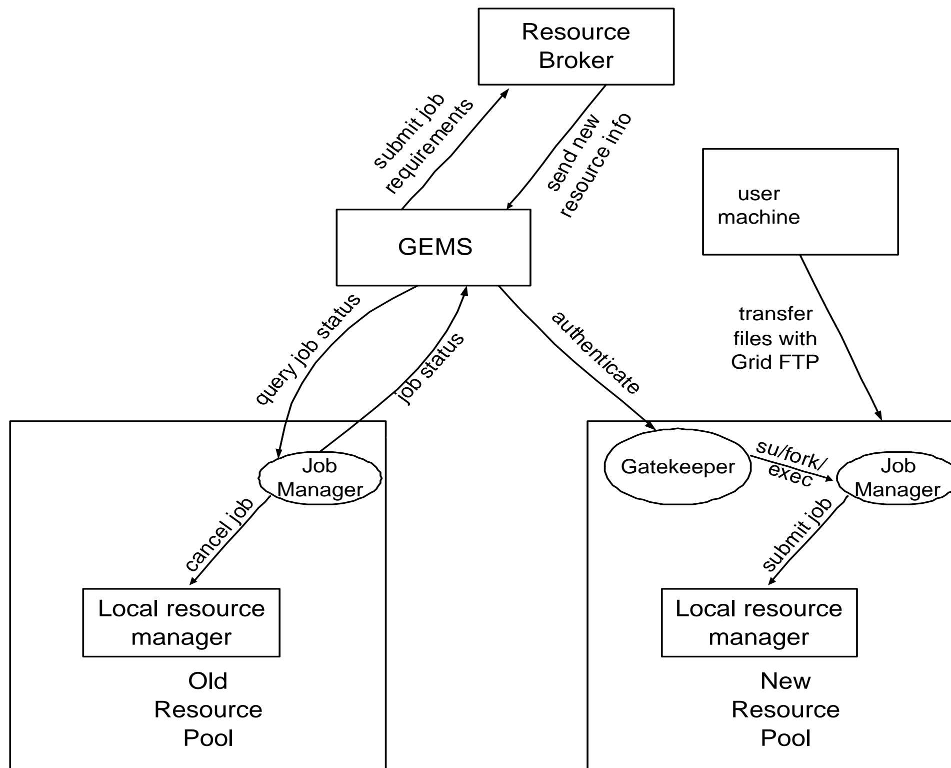

Figure 4.9: Job migration across resource pools using GEMS

gatekeeper. The Grid FTP module then transfers the input files and executables from the user's machine to the new resource. A new job manager is started and it submits the job to the DQ scheduling system of the new resource. GEMS periodically contacts this new job manager to query the status of job. Once the job completes its execution, GEMS transfers the output files, input files and executables back to the user's machine. The job migration also helps to decrease the load on the older resource pool, as it releases the resources allocated to the job that remain unused as the job is in restart state. So another job which requires fewer resources can be started. GEMS need not collect monitoring information at very short intervals. So this does not lead to excessive monitoring information traffic across the Grid.

Chapter 5

Implementation Issues

5.1 Implementation of DQ Queuing System

The DQ queuing system is designed by Dr. Srinidhi Varadarajan. He and Ms. Prachi Bora have set up the initial data structures and implemented the scheduler thread.

DQ consists of a server program which resides on a designated machine in the resource pool. A DQ monitor client runs on all the other machines in the resource pool. The users connect to the DQ server using a client program called "nq". For example, in the case of a cluster, the DQ server runs on the head node and monitor clients run on all the other nodes of the cluster.

5.1.1 DQ Server Data Structures

The DQ server maintains information about the queues, machines, submitted jobs, jobs to be scheduled and jobs to be restarted. For each of these entities, a linked list is created containing pointers to the actual record. The information stored in the record for each of these is discussed below.

- **Queues:** The queue record contains the name of the queue, a state variable which specifies whether the queue is on-line or offline and a linked list of pointers to the records of the machines which support the queue. A given machine may support more than one queue.

- **Machines:** The record for a machine contains the name of the machine, maximum job processes that can be scheduled on it, the number of current job processes, a state variable that

specifies whether the machine is on-line or offline, a record that holds the information sent by the monitor client of the machine, a linked list of pointers to the records of the jobs running on this node and a linked list of pointers to the queue records that the machine supports.

- **Jobs:** The job record contains information about the job: number of machines needed, the queue type of the job, the id of the user who submitted the job, the working directory, executable name, arguments, job id assigned, status of the job ("waiting", "running", "restart" or "done"), job type (MPI or not), the number of job processes that finished execution successfully and a linked list of pointers to the machine records allocated to the job.

Linked lists of pointers are maintained for the queue records, job records and machine records. In addition linked lists of pointers to the job records that are waiting to be scheduled by the scheduler thread, and failed jobs that are waiting to be restarted by the restart scheduler thread are also maintained. So there is a job list containing pointers to all job records, a waiting job list containing pointers to job records that are to be allocated resources by the scheduler thread and a failed job list that contains pointers to the jobs that are waiting for the restart scheduler thread to allocate resources for restart. Most of the DQ system functions require the record of a specific job, queue or machine. Traversing the linked lists linearly to retrieve a particular record is not efficient if we have a huge resource pool. So hash tables are maintained for these records. A pointer to the record is stored in a hash table and can be accessed with a key. The key for the queue records is the queue name, for job records it is the job id, and for machine records it is the machine name.

5.1.2 DQ Functionality

Recall from Chapter 4 that DQ consists of two main components – server and monitor clients. The DQ server consists of the following threads

- DQ server thread: Accepts requests from users.
- Scheduler: Allocates the machines in resource pool to jobs.
- Job startup: Responsible for starting the scheduled jobs.
- Monitor server: Receives monitoring information form monitor clients on various machines.

- Monitor timer: Checks if all the monitor clients have sent messages
- Restart Scheduler: Allocates the machines in restart pool to the failed jobs.
- Job restarter: Responsible for restarting failed jobs once the restart scheduler assigns them a machine.

System Startup

DQ is a multi-threaded server implemented using the PThreads library. When the DQ server starts it reads in the configuration files *queue.conf* and *node.conf*. The *queue.conf* file contains a description of the queues to be supported. The entries in this file are of the format "queue name : description of queue". The *node.conf* file contains the description of the machines in the resource pool. The entries in this file are of the format "machine name: number of processes supported: queues supported". Example *queue.conf* and *node.conf* files are shown in Figure 5.1. These files are parsed and the information is stored in corresponding queue and machine records.

```
# Example queue.conf file

prod_q: production
exp_q: experimental
restart_q: restart
```

```
#Example node.conf file

anantham6.cs.vt.edu:  1: exp_q: prod_q
anantham8.cs.vt.edu:  1: exp_q: prod_q
```

Figure 5.1: Example queue and node configuration files

User Interaction

The users connect to DQ using a client program called "nq" which can be used with different options for job submission, querying status of jobs and also to cancel jobs. Some of the nq options are listed in Figure 5.2.

Job Submission and Start up

When the user submits a job, DQ parses the job submission request and stores the information in a job record. It sets the job state as "waiting" and adds pointers to the record in the job linked list, job hash table and the waiting job list. A job id is returned to the user. DQ then wakes up

```
Job Submission: nq -q queue name -n no. of nodes -m -w workingdir executable arguments
    -q queue name: specifies the queue type of the job
    -n no. of nodes: specifies the number of machines required for the job
    -m: specifies that job is a MPI job
    -w : working directory

Job Status: nq -l [jobid]
    -l: lists status of jobs in all queues. With job id gives status of the particular job

Job cancel: nq -d jobid
    -d jobid: cancels a job

List of nodes executing a particular job: nq -nodes jobid

Jobs in a particular queue: nq -queue “queue name” -l

Nodes in a particular queue: nq -queue “queue name” -n

Queues supported by a particular node: nq -node “node name” -q

List of offline nodes: nq -node -off

Number of free machines: nq -node -f
```

Figure 5.2: nq options

the scheduler thread. The scheduling algorithm currently implemented is a “first come first served” algorithm. So the scheduler thread starts at the head of the waiting jobs list and allocates the free machines to the jobs until it finds the first job whose requirement cannot be satisfied with the current free machines in the resource pool. At this point the scheduler thread goes back to a conditional wait state again. The monitor thread wakes it up again whenever machines become available as explained later. In the future implementation more sophisticated scheduling algorithms taking job priorities into consideration will be used.

For each job record that the scheduler thread retrieves from the waiting jobs list, it reads the queue type of job and retrieves the corresponding queue record using the queue hash table. As explained above, the queue record contains a linked list of pointers to the machine records that support the queue. The scheduler then checks if enough machines are available that support the desired queue. This is done by going through the machine records list in the queue record looking for the machines that are on-line and with fewer than the maximum number of processes running.

If there are enough machines, the scheduler allocates them to the job by placing pointers to the machine records in the machine list of the job record and changing the job status to “running”. When the scheduler thread finds the first job that cannot be scheduled, it wakes up the job startup thread and goes back to a conditional wait state.

The job startup thread goes through the waiting jobs list and checks the status of each job. If a job is marked “running”, the startup thread goes through the machine list in the job record and sends a message to the monitor client on each machine to startup the job. The job startup thread then removes the job pointer from the waiting jobs list. The job startup thread goes back to a conditional wait state as soon as it finds the first job in the waiting jobs list whose status is “waiting”.

The job startup message that the monitor clients receive consists of the job executable name, arguments and the job id. The monitor clients maintain information about the jobs running on their machine in a linked list. Each monitor client parses the job startup message, stores the information in a new job record and adds it to the local job linked list. The monitor clients startup the job process using the *exec* system call. Before that, the standard out and standard error of the process is set to be directed to the files $< job > . < jobid > o$ and $< job > . < jobid > e$ respectively where $< job >$ is the name of the user's executable. Also the process owner is set to the user id. MPI job processes require some special environment variables for their startup. The monitor clients set these environment variables before MPI job process startup. If the job startup fails, the monitor clients report this to the monitor server which takes care of it as explained below. The job startup thread sends the job startup message to the monitor clients using a TCP connection. Since TCP is a reliable protocol, it guarantees that the job startup message reaches the monitor client.

Fault Detection

The DQ monitoring system consists of a monitor server, a timer and monitor clients. The monitor server and timer are implemented as threads in the main DQ server and so they reside on the front end node of the cluster. The monitor clients reside on every machine of the cluster. The monitor clients send two kinds of messages to the monitor server:

- Machine status: The machine status message is sent periodically to the monitor server by all the monitor clients. It consists of the machine name along with information about the machine such as current free memory and CPU load. The monitor client can obtain this information

by querying */proc* of the machine. This machine status message also serves as a heartbeat message through which the monitor server comes to know that the corresponding machine is on-line. When the monitor server receives a machine status, it retrieves the corresponding machine record by using the machine name as key to the machine hash table. It updates the information stored in the record using the new message and sets a machine state variable to *on* that shows that the machine is on-line. It also sets a counter in the machine record that counts the number of missing heartbeats to zero. The monitoring information is used by the system administrators for monitoring the resource pool.

- Job status: Whenever a job process executing on a machine exits, the monitor client sends a job status message to the monitor server. The message contains the job id and a job status variable that specifies if the job has exited normally or due to some fault. This message is used by the monitor server to decide if the job is done or has failed.

The monitor clients communicate with the monitor server using UDP messages. This is preferred over TCP because TCP is a connection oriented protocol that consumes resources on both the sender and receiver; the overhead associated with UDP messages is much less. Heartbeat messages are sent to the monitor server periodically by all the monitor clients. In a large resource pool with hundreds of machines, implementing the heartbeat as TCP messages will lead to excessive load on the monitor server. So for scalability reasons heartbeats have been implemented as UDP messages.

The DQ monitoring system distinguishes between two kinds of failures for MPI jobs:

- Hardware failures: This kind of failure occurs when the machine on which the job process is running goes down, thereby killing the job process. Any other kind of failures on the machine (such as network card failure) which prevent the MPI process from communicating with peers also come under this category. Hardware failures are detected through missing heartbeat messages. The timer thread periodically goes through all the machine records and checks if all the monitor clients have sent heartbeat messages. In the current implementation, the timer thread checks all the machine records every 5 minutes. The interval for the timer can be set according to the applications running in the resource pool. For long running jobs a longer timeout can be used. The value is set by changing the timeout value in a configuration file.

  When the timer thread wakes up after time out, it checks if a heartbeat for each machine has been received during the timeout period. This is done by checking if the machine state

variable is *on*. If the variable is *on*, the timer sets it to *off*. Later when the monitor thread receives a heartbeat message it turns the variable *on*. If the timer thread finds the variable *off*, it means that the monitor server did not receive the heartbeat message from the monitor client during the timeout period. The timer takes a note of this missing heartbeat by incrementing the missing heartbeat counter. If the timer thread finds the machine state variable *off* in its subsequent periodic iterations too, it means that the heartbeat messages are continuously missing from the machine. The timer records this by incrementing the missing heartbeat count. In the current implementation, the timer thread decides that a machine has gone down when the missing heartbeat counter in the machine's record is 3. The timer does not decide about the status of a machine based on a single missing heartbeat. This is because heartbeats are UDP messages and they are unreliable. But the probability of a UDP message being continuously dropped is very low. That is why the timer thread checks if the heartbeats are continuously missing by using the missing heartbeat counter value. If the monitor server does not receive the heartbeat from a machine in 3 continuous timeout periods, the timer thread would have incremented the missing heartbeat count to 3. If a heartbeat is received in between, say after 2 timeout periods, the monitor server resets the missing heartbeat count to zero and sets the machine state variable *on*. Once the timer thread assumes that the machine has gone down, it initiates the job restart process for the jobs running on that machine. The job restart process is explained below.

- Software failure: This kind of failure occurs if the job process exits due to a bug in the application code such as a divide by zero error. The job startup mechanism used in DQ helps to detect such failures. As explained earlier, a job processes is started by the monitor client using the *exec* system call. This makes the monitor client parent of the job process. The parent receives a `SIGCHLD` signal when its child exits. The monitor client has a signal handler to handle this signal. The signal handler uses the *waitpid* system call to determine the pid of the exited job process. The system call can also be used to determine if the job process has gracefully finished execution and exited or exited due to some fault. The monitor client then sends this information (job id of the job process exited and the reason for exit) to the monitor server in a job status message. Job startup fails if the *exec* system call used to start up the job fails. This can also be detected by the monitor client and reported to the monitor server. This kind of job control is not possible through mechanisms which use rsh or ssh for

job startup. In such mechanisms the job startup program on the head node uses *rsh* or *ssh* to send the job startup message to every machine allocated to the job. Thus, the job startup program is not the parent of any of the job processes as they are on different machines; hence it has no knowledge of the job process execution. So a finer level of job process monitoring is made possible by making the monitor client the parent of the job process.

Job Restart

The DQ monitoring system can detect hardware as well as software failures and report them to the monitor server. DQ tries to restart the job in case of a hardware failure. In case of a software failure the job is terminated as it is assumed that software failures may be due to logical errors in the application code which would be repeated even after a restart.

When a job fails due to a software failure, the monitor server comes to know of it through the job status message sent by the monitor client. The monitor server then terminates the job by sending a job cancel request to the monitor clients of all the machines allocated to the job. The monitor clients in turn kill the job processes on their machines. The monitor server releases the resources allocated to the job and writes the reason for termination to a job log file named $< job >< jobid >$ *.log*.

When a job fails due to a hardware failure, DQ tries to restart it by replacing the failed machines with new machines obtained from a restart pool. The restart pool consists of the machines that support the restart queue as defined in the node configuration file. As explained above, the timer thread detects the machine failures through missing heartbeats. When the missing heartbeats counter reaches a fixed value, the timer decides that the machine has gone down and sets a flag in the machine record that shows that the machine is offline. When a machine fails, all the jobs whose processes were running on that machine have to be killed. The timer thread goes through the job list of the failed machine. This list contains pointers to the records of jobs that were running on that machine. Each job record in-turn contains the list of machines allocated to it. The timer thread removes the failed machine from this list and sends a job cancel message to the monitor clients of all the other machines. The monitor clients in-turn kill the job processes. For each cancelled job, the job state is changed to "restart" and a pointer to this job is added to the list of jobs waiting in the restart queue. The timer thread then wakes up the restart scheduler.

The restart scheduler retrieves the restart queue record using the queue hash table. This record contains a linked list of pointers to the machines that support the restart queue. The restart

scheduler then goes through the list of jobs waiting in the restart queue. Each job in the restart queue requires a single machine on which the failed job process can be restarted as they retain all the other allocated machines. So the restart scheduler checks if a machine is available in the restart pool that can accommodate a single job process. If so, it allocates the machine to the job by placing a pointer to the machine record in the machine list of the job record and then changes the state of the job to "running". As soon as the restart scheduler finds the first job that cannot be scheduled due to lack of free machines, it wakes up the job restart thread and goes back to a conditional wait state until free machines become available. The job restart thread goes through the list of the failed jobs in the restart queue and for each job checks if the job status has been changed to "running". If so it sends a job start request to all the machines that have been allocated to the job and removes it from the restart queue. As soon as the job restart thread finds the first job that is in "restart" state, it goes back to a conditional wait state. In this way, the failed machine is replaced with a new one and the job is restarted.

As soon as the timer thread detects a machine failure it submits the jobs on that machine to a restart queue as explained above. All the machines except the failed one allocated to the job are retained. So when a job is submitted to the restart queue, only one machine is required to restart the job. It is possible that some other machines allocated to the job fail as it waits in the restart queue. In this case, the job restart will fail when the job restart thread tries to send a job start request to the failed machine. The timer thread will eventually detect the failed machine and initiates the job restart process.

Job Completion

As already explained, the job status message sent by a monitor client contains the id of the job and the reason for the job exit. The monitor server retrieves the job record using the job id as key to the job hash table. If the job process has exited gracefully, the monitor server increments the field in which the number of job processes that finished execution is stored. If the count equals the number of nodes in the job request then it means that all job processes have finished execution successfully. If so the monitor server releases all the resources allocated to the job. This is done by going through the machine records of the job and decrementing the field containing the count of current job processes on the machine. If the job status message says that the job process failed due to some problem, the monitor server writes a message to a job log file $< job >< jobid > .log$

and terminates the job by sending a job cancel message to all the monitor clients and releases the machines allocated to the job. In any case both the scheduler thread and restart scheduler thread are woken up because machines become available.

The various threads in DQ perform operations on a number of shared data structures like the machine lists, job lists, etc. Access to these data structures is controlled through mutexes and care has been taken so that the threads do not end up in a dead lock. All the threads in DQ wait on a condition instead of using the expensive polling mechanisms. An entire thread can be easily replaced by another with better functionality. This requires minimum changes in the source code.

DQ Monitor Client

The DQ monitor client performs a number of activities as explained above. The monitor client is also multi-threaded. The main thread thread listens on a port for job startup requests. It is the parent of all the job processes that the monitor client starts up and receives a `SIGCHLD` signal whenever a job process exits. So it has a signal handler that checks the exit code of the job and sends a job status message to the monitor server as explained above. The other thread in the monitor client periodically (3 minutes) sends the machine status message which also serves as a heart beat to the monitor server as explained above.

In the current implementation, when the DQ server starts up, it starts up the monitor clients on all the machines using rsh. In the future implementation, the monitor clients will be placed in the machine startup.

5.2 Implementation of GEMS

GEMS is used for submission, monitoring and management of jobs to Grid resources. GEMS consists of a server that runs on a designated machine and a client program that users can use from their machine to connect to the server and perform various tasks. GEMS is implemented using the Java CoG Kit. The GEMS server consists of the modules as shown in Figure 5.3. The following sections explain these modules.

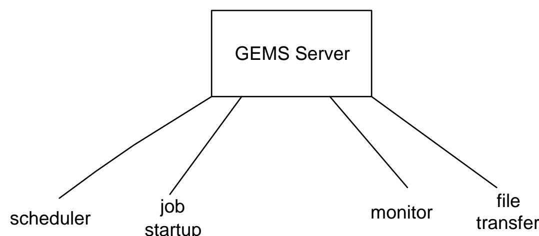

Figure 5.3: GEMS modules

5.2.1 Job Submission

The user describes a job request in a file using the Simulation Definition Language (SDL). SDL provides a XML-based representation of simulations or jobs. The core set of tags can be extended for additional functionalities. It is difficult to express all the user's needs by the options in a RSL. SDL provides a convenient way to describe a job request in XML format. Also the user request may be passed to a resource broker to find the best suitable resource. It is convenient to pass a XML file to the broker than a RSL string. The functionality for parsing this file has to be implemented in the broker. Figure 5.4 shows a sample SDL file.

The SDL schema allows a user to define the executable, input and output file names and system requirements for the job. These system requirements will be used by a resource broker to find a suitable resource for the job. The semantics of the tags are given in Tables 5.1 and 5.2.

XML Tag	**Semantics of the Tag**
no_procs	number of processors
working_directory	working directory where executable can be found on the user's machine
executable	name of the executable
stdin	standard input for the job
stdout	standard output for the job
input_files	input files for the executable
output_files	output files for the executable
cpufree	average cpu utilization for a period of 1,5 or 15 minutes
fsfree(in byte)	amount of free file system space
ramfree(in bytes)	amount of free memory
vmfree(in byte)	amount of free virtual memory
osname	name of operating system
osrelease	release version of the operating system
rank	system rank attribute

Table 5.1: SDL tags

```
<?xml version="1.0" encoding="UTF-8"?>
<simulation>
        <host>sachem.cs.vt.edu</host>
        <no_procs>2</no_procs>
        <working_directory>/home/stadepal</working_directory>
        <executable>test</executable>
        <job_type>mpi</job_type>
        <stdout>OUT</stdout>
        <input_files nfiles="2">
                <file>/home/stadepal/data/in1</file>
                <file>/home/stadepal/data/in2</file>
        </input_files>
        <cpufree average="1min" equality="greaterthan">10</cpufree>
        <fsfree equality="greaterthan">10000</fsfree>
        <ramfree equality="greaterthan">10</ramfree>
        <vmfree equality="greaterthan">40</vmfree>
        <osname>Linux</osname>
        <rank>cpufree</rank>
</simulation>
```

Figure 5.4: Sample SDL file

When the user wants to submit a job, he stores his proxy credentials used to log into Grid resources on a MyProxy server. Then he starts the GEMS client. The client prompts the user for the location of the SDL file, the user name and password used for MyProxy server. The client retrieves the SDL file and transfers it to the GEMS server along with the user name and password. The GEMS client can later be used to check the status of jobs. Additional functionality like killing a submitted job and initiating job migration through the GEMS client will be implemented in the future.

The server parses the SDL file using Xerces XML parser, stores the information in a linked list of jobs and marks the state of the job as "unsubmitted". The scheduler module is implemented as a thread that wakes up periodically after a timeout (10 minutes in the current implementation). In the current implementation the details about the available resource pools (the hostname to be contacted) are stored in a file. Later the scheduler will be interfaced with a resource broker. The scheduler assumes that the local queuing system at each resource is DQ. For each resource the

XML Attribute	Semantics of the attribute
nfiles	number of nested tags for a multivalued tag
equality	Relational operator to be used in conjunction with element
average	specific to cpu utilization. It can hold only 3 possible values of 1,5 or 15 minute averages

Table 5.2: SDL attributes

scheduler gathers information about the number of free machines. This is done by submitting a command to DQ (nq -node -f). Once the scheduler has details about free machines in each resource, it goes through the job list. For each job in "unsubmitted" state, the scheduler checks if any of the resources can accommodate the job (machines required by the job is less than free machines in the resource pool). If so, the scheduler invokes the job submission module with the job record and resource pool contact (the hostname of the machine to contact). If no resource is available the job waits in the queue.

The job submission module of GEMS uses the classes provided in org.globus.myproxy and org.globus.gram packages in the Java CoG kit. The classes in myproxy package are used to retrieve the proxy credentials from a MyProxy server and those in gram package are used to submit the job and find its status. The job submission module contacts the MyProxy server with the user id and password and retrieves the proxy credentials of the user. As explained in Chapter 2, the Globus Resource Allocation Manager (GRAM) contains a gatekeeper component on each Grid resource that performs mutual authentication and a job manager that serves as the job contact. The job submission module contacts the gatekeeper of the resource pool and performs the process of authentication using the proxy credentials. The file transfer module of GEMS uses the classes provided by org.globus.ftp package in the Java CoG kit to do a third party transfer of the input files and executables from the user's machine to the resource. Once the files are transferred, the job submission module forms a RSL of the job using information in the job record that was obtained from the job SDL. It then submits the job to the gatekeeper of the resource. This is done through the classes provided in gram package of Java CoG kit. The gatekeeper starts a job manager for this job. To interface DQ with the GRAM job manager a PERL module is written. Separate PERL scripts for submitting a job to DQ, querying the status of a job and canceling a job are written. The job manager executes one of the scripts depending on the request it receives. The PERL script translates the request to the appropriate DQ command, submits it to DQ and returns the output to the job manager. In the case of job submission, the job manager executes the job submission PERL

script and passes it the options in the RSL of the job. The PERL script forms the appropriate DQ command and submits the job to DQ. It returns the job id returned by DQ to the job manager. The job manager forms a job URL using this job id and returns it to the job submission module. The job submission module places the URL and the resource pool contact (hostname) in the job record. The state of the job is changed to "pending".

5.2.2 Job Monitoring

The job monitoring module in GEMS is also implemented as a thread that periodically wakes up after a timeout (10 minutes in current implementation). The job monitor goes through the list of jobs and for each job that is not in "unsubmitted" state, it finds the information about the resource on which the job is currently running from the job record. The job monitor uses the classes in gram package of Java CoG kit. It contacts the corresponding resource's job manager with the job URL. The job manager executes the job status PERL script to find the state of the job and returns it to the monitor. The monitor updates the state in the job record. Thus the job state in the resource is propagated up to the GEMS server.

Figure 5.5 shows the typical states which a job goes through on the Grid when submitted through GRAM.

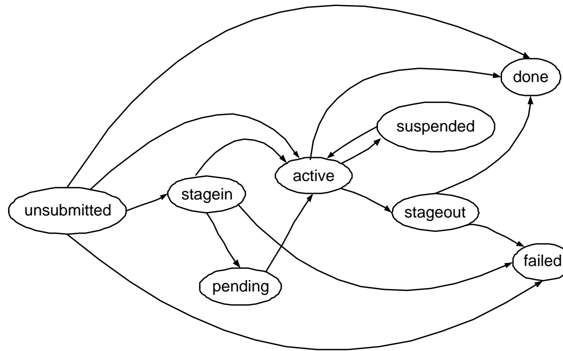

Figure 5.5: GRAM job states

When the job is submitted through GRAM to a resource pool managed by DQ, the job state machine would be as shown in Figure 5.6. The "restart" state is added because DQ supports the restart of failed jobs, unlike many other queuing systems. So the job manager for DQ must keep track of these states and report them back to GEMS.

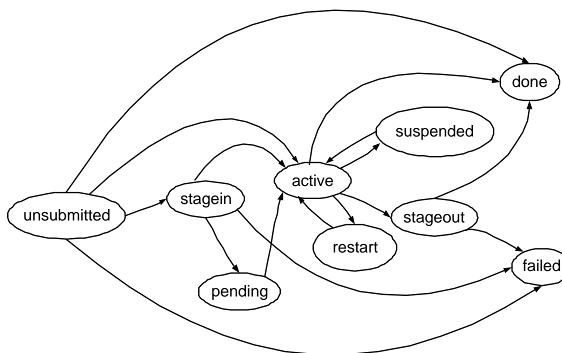

Figure 5.6: GRAM job states with DQ

5.2.3 Job Migration

As the job execution goes on, some hardware failure in the resource pool might occur causing one of the job processes to die. DQ then places the job in its restart queue. The GEMS job monitor also comes to know that the job is in "restart state". If all the machines in the restart pool are used up, this job might not be restarted immediately. If the load on the system is high, the job could remain in the restart queue for a long period. The GEMS job monitor detects this with the help of a counter. Whenever the job monitor wakes up after a time out and finds a job in "restart" state, it increments a counter in the job record. If the counter reaches a value of 3, that is if the job is still in restart state even after the GEMS job monitor has gone through 3 timeouts, the GEMS monitor tries to migrate the job to some other resource pool where job execution can be started immediately. The GEMS monitor wakes up the scheduler thread which tries to find a new resource as described above. If a new resource is found the monitor sends a job cancel request to the old resource through the job manager. The executables and input files are transferred to the new resource from the user's machine using the file transfer module of GEMS. The new resource's gatekeeper is contacted for authentication and it starts a new job manager for the job. The job state and resource information are updated in the job record. Once job execution is finished the output files are transferred back to the user's machine using Grid FTP.

Chapter 6

Conclusions and Future Work

Job checkpointing and migration is very important to utilize the true power of Grid infrastructures. This thesis is an effort to provide user transparent job migration across Grid resources by utilizing the fault tolerant capabilities of local queuing systems.

6.1 Conclusions

We have developed a prototype implementation of the fault tolerant queuing system (DQ) and Grid job management system (GEMS) as described in the previous chapters. The prototype GEMS accepts job requests from users and submits jobs to resource pools managed by local DQ schedulers. DQ detects job process failures and restarts the job by obtaining a new machine from a restart pool. If no machines are available in the restart pool, DQ suspends the job. GEMS monitors the status of the job by periodically querying the job manager. Thus, GEMS can detect that the job is suspended due to failure. Then GEMS tries to find a new resource where the job can be started immediately. If it finds such a resource pool, it moves the job to the new resource.

The prototype implementations demonstrate the fault detection capabilities described in previous chapters. GEMS uses the fault detection capabilities of DQ to provide transparent job migration across grid resources. These prototypes demonstrate that job migration across Grid resources can be achieved if the local queuing system handles the fault detection tasks and reports them to the Grid job management system. This results in effective utilization of resources.

6.2 Future Work

In the prototype implementation of DQ, the scheduling algorithm implemented is a simple "first come first served" algorithm. The emphasis has been on the monitoring system which is completely in place. A more sophisticated scheduling algorithm taking job priorities into consideration has to be implemented. Time limits for the jobs are to be enforced and an accounting policy also needs to be integrated. The job restart mechanism will be interfaced with Déjà vu to restart failed jobs from a checkpoint. The entire DQ system - server, various queues and client processes will be checkpointed using Déjà vu to protect them from failures. Also the standard output and standard error for the job processes needs to be set correctly. Methods for utilizing the monitoring system of DQ with any kind of queuing system to detect machine failures are to be devised.

Regarding GEMS, the current implementation accepts a job request, schedules it on one of the available resource pools, monitors the job processes for failures, migrates the job in case of failure and finally when the job execution is complete, transfers the output back to the user's machine. The users cannot cancel a submitted job or initiate a migration event by themselves. GEMS has to be enhanced with these functionalities. Also GEMS has to be interfaced with a resource broker for efficient resource discovery taking job requirements into consideration. In the current implementation, the user has no control over job migration. The users might want to run their jobs on a particular set of resources. Even if a failure occurs, the jobs have to be migrated only between these resources. The users can specify such requirements in the SDL file and GEMS must be able to enforce them. The users can also specify different types of executables for different machine architectures. GEMS must be able to choose the correct executable depending on the resource it obtains. Another possible improvement is credential management. The proxy credential of a user expires after a certain period. If the user's job is not completed within this period, GEMS can warn the user through email. The current implementation is based on Globus 2. Using Globus 3 GEMS can be implemented as a web service which is more usable.

The prototype GEMS restarts a job on the new resource from scratch. As the Déjà vu checkpointing system evolves, GEMS will utilize it to restart the job from a checkpoint on a new resource. The process of transferring the required input files and checkpoint files from databases will be handled by GEMS. The users can send a job checkpoint signal through GEMS and initiate migration. The GEMS server will also be checkpointed using the checkpointing algorithms of Déjà vu to protect it from the failure of the machine on which it runs. Sophisticated migration strategies that take the

network bandwidth between resources and size of files to be transferred need to be implemented in GEMS. The current implementation has a naive migration strategy that migrates a failed job after a fixed amount of time. There are several improvements possible for the job submission strategy used by GEMS. In the current implementation the jobs remain in the GEMS queue if they cannot be submitted to any resource pool. In this case, the job can be submitted to the pool in which resources will be available as soon as possible. The job waits in the queue in that pool and will be scheduled as soon as resources are available.

Integrating GEMS with Déjà vu makes the finest level of migration possible across the Grid – job process migration. In the current implementation, when GEMS submits a MPI job to a resource pool, all the job processes reside in the same pool. Déjà vu's concept of virtual network allows MPI job processes across various resource pools to communicate and still utilize its checkpointing abilities unlike MPICH–G2. So GEMS can schedule the job processes on different resource pools. When a job process fails, only that job process can be migrated to another resource pool instead of migrating the entire job as in the current implementation. This kind of job process migration across the Grid allows load balancing parallel applications over the Grid resources. Whenever the load on a resource pool becomes high, certain number of processes can be checkpointed and moved to another resource pool through GEMS.

Considering the scalability of GEMS, we can imagine one GEMS per each organization. If there are many resources and large number of jobs possible, then more than one GEMS per organization is possible. GEMS can submit a job to any resource which the resource broker finds. So only the number of jobs is the factor that decides the number of GEMS needed.

REFERENCES

[1] A. Agbaria and R. Friedman. Starfish: Fault-tolerant dynamic mpi programs on clusters of workstations. In *8th IEEE International Symposium on High Performance Distributed Computing*, 1999.

[2] W. Allcock, J. Bester, J. Bresnahan, A. Chervenak, L. Liming, S. Meder, and S. Tuecke. Gridftp protocol specification. In *GGF GridFTP Working Group Document*, September 2002.

[3] R. Batchu, J. Neelamegam, Z. Cui, M. Beddhua, A. Skjellum, Y. Dandass, and M. Apte. Mpi/ft: Architecture and taxonomies for fault-tolerant, message-passing middleware for performance-portable parallel computing. In *Proceedings of the 1st IEEE International Symposium of Cluster Com- puting*, 2001.

[4] J. Bester, I. Foster, C. Kesselman, J. Tedesco, and S. Tuecke. Gass: A data movement and access service for wide area computing systems. In *Sixth Workshop on I/O in Parallel and Distributed Systems*, 1999.

[5] K. Birman. The process group approach to reliable distributed computing. In *Communications of the ACM*, 1993.

[6] G. Burns, R. Daoud, and J. Vaigl. LAM: An Open Cluster Environment for MPI. In *Proceedings of Supercomputing Symposium*, pages 379–386, 1994.

[7] K. Czajkowski, S. Fitzgerald, I. Foster, and C. Kesselman. Grid information services for distributed resource sharing. In *Proceedings of the Tenth IEEE International Symposium on High-Performance Distributed Computing (HPDC-10)*. IEEE Press, August 2001.

[8] K. Czajkowski, I. Foster, N. Karonis, C. Kesselman, S. Martin, W. Smith, and S. Tuecke. A resource management architecture for metacomputing systems. In *Proc. IPPS/SPDP '98 Workshop on Job Scheduling Strategies for Parallel Processing*, pages 62–82, 1998.

[9] G. Bosilca et.al. Mpich-v: Toward a scalable fault tolerant mpi for volatile nodes. In *Super Computing 2002*, 2002.

[10] G. Fagg and J. Dongarra. Ft-mpi: Fault tolerant mpi, supporting dynamic applications in a dynamic world. In *Euro PVM/MPI User's Group Meeting*, 2000.

[11] I. Foster, C. Kesselman, G. Tsudik, and S. Tuecke. A security architecture for computational grids. In *Proc. 5th ACM Conference on Computer and Communications Security Conference*, pages 83–92, 1998.

[12] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *International J. Supercomputer Applications*, 2001.

[13] J. Frey, T. Tannenbaum, I. Foster, M. Livny, and S. Tuecke. Condor-g: A computation management agent for multi-institutional grids. In *Proceedings of the Tenth IEEE Symposium on High Performance Distributed Computing (HPDC10), San Francisco, California*, August 2001.

[14] W. Gropp and E. Lusk. Fault tolerance in mpi programs.

[15] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, September 1996.

[16] A. Karnik and C. J. Ribbens. Data and activity representation for grid computing. Technical Report 02-13, Department of Computer Science, Virginia Polytechnic Institute & State University, Blacksburg, VA, 2002.

[17] N. Karonis, B. Toonen, and I. Foster. Mpich-g2: A grid-enabled implementation of the message passing interface. *Journal of Parallel and Distributed Computing*, 2003.

[18] G. Laszewski, I. Foster, J. Gawor, and P. Lane. A java commodity grid kit. *Concurrency and Computation: Practice and Experience*, 13(8-9):643–662, 2003. http://www.cogkits.org/.

[19] M. Litzkow, M. Livny, and M. Mutka. Condor—a hunter of idle workstations. In *Proceedings of the 8th International Conference on Distributed Computing Systems*, pages 104–111, Los Alamitos, CA, 1988. IEEE Computer Society Press.

[20] S. Louca, N. Neophytou, A. Lachanas, and P. Evripidou. Mpi-ft: Portable fault tolerance scheme for mpi. In *Parallel Processing Letters*, 2000.

[21] J. Novotny. The grid portal development kit. In *Concurrency: Pract. Exper.*, 2000.

[22] J. Novotny, S. Tuecke, and V. Welch. An online credential repository for the grid: Myproxy. In *Proceedings of the Tenth International Symposium on High Performance Distributed Computing (HPDC-10)*. IEEE Press, August 2001.

[23] S. Rao, L. Alvisi, and H. Vin. Egida: An extensible toolkit for low-overhead fault- tolerance. In *In Symposium on Fault-Tolerant Comput- ing*, page 4855, 1999.

[24] G. Stellner. Cocheck: Checkpointing and process migration for mpi. In *Proceedings of the 10th International Parallel Processing sympposium(IPPS '96)*, 1996.

[25] M. Thomas, S. Mock, J. Boisseau, M. Dahan, K. Mueller, and D. Sutton. The gridport toolkit architecture for building grid portals. In *Proceedings of the 10th IEEE Intl. Symp. on High Perf. Dist. Comp*, 2001.

[26] S. Varadarajan, J. Mukherjee, and N. Ramakrishnan. Weaves: A novel direct code execution interface for parallel high performance scientific codes. Technical report cs.dc/0205004, computing research repository (corr), Department of Computer Science, Virginia Polytechnic Institute & State University, May 2002.

VITA

Sriram Satish Tadepalli was born in Gudiwada, India on June 18, 1980. He did his schooling in Guntur. He received his B.E (Bachelor of Engineering) in Computer Science from Nagarjuna University in 2001. After this he joined the Master's program in the Computer Science Department at Virginia Polytechnic and State University in Fall 2001. At present he is a PhD cadidate in the same department. His research interests are in data mining, operating systems, and bioinformatics. He is a member of the ACM and the IEEE Computer Society.