A Framework for QoS Adaptive Grid Meta Scheduling

K. C. Nainwal, J. Lakshmi, S. K. Nandy
CAD Lab, Indian Institute of Science
Bangalore 560 012 (INDIA)
{nainwal@cadl, jlakshmi@serc, nandy@serc}.iisc.ernet.in

Ranjani Narayan, K. Varadarajan
HP India Software Operations
Bangalore 560 052 (INDIA)
{nranjani, keshavan}@india.hp.com

Abstract

Grid environment, being a collection of heterogeneous and geographically distributed resources, is prone to many kinds of failures. In order to fully realize the potential of the Grid environment, we need mechanisms that will guarantee Quality of Service, even in the face of failures.

In this paper, we introduce the notion of scheduling based on "availability", and provide a framework (AGMetS) for QoS adaptive Grid meta scheduling. AGMetS uses "availability" to adapt to the underlying resource failures, thereby ensuring QoS even in the face of failures. We provide detailed design description, and a prototype implementation of AGMetS. We further provide the experimental results of our work, showing the effectiveness of AGMetS.

1. Introduction

Grid environment, being a collection of heterogeneous and geographically distributed resources, is prone to many kinds of failures. With Grid technologies, it is possible to construct large-scale applications over the Grid [2]. However, running applications on the Grid environment poses significant challenges due to the diverse failures encountered during execution. In order to fully realize the potential of the Grid, we need a highly available Grid environment, capable of effectively masking these failures.

We envisage that in order to make Grid a highly available environment, a middleware is needed that provides the view of an available infrastructure to the applications. To this end, we relate the application QoS requirements to the "availability" of the various resources. With every host, we associate an availability index vector. The elements of this vector specify the failure probability of the individual resources. We can use the availability index of a resource to predict the resource's future availability. The availability index also gives the availability of the system as a whole by taking into consideration the interactions between various resources and the effect of the failures caused due to the malfunctions or anomalies within these interactions.

On arrival of an application to the middleware, the middleware computes the availability of different resources(necessary for application execution) from application's QoS requirements. The middleware then guarantees this availability level throughout the execution of the application. This guarantee may not be honored in case when the resource (which is being used for the computational needs of the application) shows low availability, and so becomes incapable of meeting the application's desired availability. The middleware honors this guarantee by migrating the "computation" to some other suitable resource. In order to have the above stated middleware in place, a mechanism is needed that can schedule the computations on the suitable resources, i.e. the resources that satisfy the availability ensured by the middleware. Further, this mechanism should be capable of ensuring that the availability, which the middleware has committed, is being guaranteed throughout the lifetime of the computation. For this, the mechanism should be able to continuously monitor the availability of the resource on which the computation is being performed. In case of availability falling below the assured level, it should be able to migrate the computation to other suitable resources.

In this paper, we design AGMetS framework for QoS adaptive Grid meta scheduling and provide a prototype implementation of AGMetS. The goal of designing AGMetS is to implement the above stated mechanism. We assume without loss of generality, that a method to periodically compute the availability index vector for all the hosts in the Grid is readily available. AGMetS makes intelligent scheduling decisions based on the knowledge of the availability indices of resources. AGMetS uses "availability" to adapt to the underlying resource failures. AGMetS also renders an autonomous behaviour towards resource failures, and improves the job's response time. It offers a coherent view of availability to the applications, by masking the underlying resource failures.

The rest of the paper is organized as follows. In section 2, we discuss the related work. Section 3 gives the system design description. In section 4, we provide the details of prototype implementation. Section 5 describes the experimental results and finally section 6 concludes the paper.

2. Related Work

A great deal of work has been done on Grid resource brokering, and fault tolerance in Grid environment, but none has addressed fault tolerance as a goal for brokering.

There are many projects that investigate resource brokering on Grid. In AppLes [3], authors talk about scheduling agents for individual Grid applications. These agents use application oriented scheduling, and select a set of resources taking into consideration application and resource information.

Nimrod/G [4] adopts economic theories in Grid resource management. The scheduling policy is driven by user-defined requirements such as budget/deadline limitations.

GARA [6] is a distributed resource management architecture that supports advance reservations and co-allocation. Unlike in GARA, the emphasis in AGMetS is on "availability". GARA uses resource based QoS parameters for reservation and co-allocation, and thereby guarantees QoS, while AGMetS performs dynamic scheduling based on "availability" to ensure QoS.

Our work differs from the above schemes in many ways. We view the resource availability as an internal translation of application's QoS parameters, and take into consideration this availability while making scheduling decisions. All the previous work perform static scheduling, which does not tolerate resource failures, whereas AGMetS does dynamic scheduling to tolerate resource failures. Lastly, AGMetS acts autonomously towards failures, thereby improving the job's response time.

Similarly, there are many techniques proposed for fault tolerance in Grid environment. There are techniques such as Retrying, Replication and Checkpointing, that can be applied at task level to handle failures.

Retrying is the simplest failure recovery technique. The motivation behind retrying is that in most cases, the cause of failures is not encountered in subsequent retries, and so the job may successfully complete its execution in a finite number of attempts. While being simple, this technique is not very effective. If the same problem persists every time, retrying is not effective anymore.

The basic idea in replication is to have replicas of a task run on different Grid resources, so that as long as not all replicated tasks crash, the task execution succeeds. The problem with this approach is that it consumes more resources than that required by the task.

Primary-Backup [5] is another failure handling technique for Grid services. A client sends a request to the primary service, which receives and services the request. The primary service then sends a state update message to the backup services, and sends replies to the client. Primary-backup requires that a client be notified of primary service failure and be allowed to rebound to the newly-appointed primary service. In this approach, many similar processes need to be carried out in parallel, resulting in low throughput. Another drawback is that the need for keeping backups consistent with the primary, incurs high overheads.

Checkpointing saves the state of a running process, so that in case of failures, the process can be started from its last checkpoint. This saves the overhead of restarting it all over again. We make use of checkpointing in our work, but instead of maintaining periodic checkpoints (as its done in most cases), we checkpoint our process only when the availability of the system goes down. This saves the overheads associated with periodic checkpointing.

Our scheme is proactive, rather than reactive, in nature. We take appropriate actions to make the resource failures transparent to the application, even before that fault actually occurs. The novelty of this scheme comes from the fact, that it addresses both issues together- resource brokering as well as fault tolerance.

3. System Design Description

In section 1, we talk about a middleware that maintains the view of an available Grid infrastructure. When an application request comes, the upper layers in this middleware compute required availability from application's QoS parameters. This availability, along with application's other requirements, is given to AGMetS. AGMetS collects resource information of all the hosts in the Grid, and matches them against the application's requirements to find the most suitable host. It submits the application to this host and continuously monitors the availability of this host till the application completes its execution. If the host's availability falls below the required level, AGMetS migrates the application to a suitable host. AGMetS thus makes sure that the resource failures are always masked from applications.

3.1. Overall Architecture

The main components of AGMetS, as shown in Figure 1, are User Interface (UI), Information Aggregation Unit (IAU), Availability Conscious Resource Management Unit (ACRMU) and Job Submission Unit (JSU).

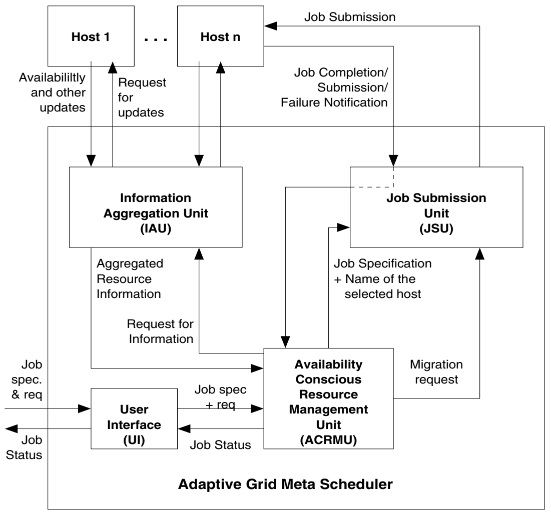
Host 1
. . .
Host n
Job Submission
Availabililtly and other updates
Request for updates
Job Completion/ Submission/ Failure Notification
Information Aggregation Unit (IAU)
Job Submission Unit (JSU)
Aggregated Resource Information
Job Specification + Name of the selected host
Request for Information
Job spec. & req
Job spec + req
Availability Conscious Resource Management Unit (ACRMU)
Migration request
User Interface (UI)
Job Status
Job Status
Adaptive Grid Meta Scheduler

Figure 1. Overall Architecture of AGMetS.

UI acts as an interface between the user and rest of the system. IAU is responsible for gathering the resource information of the hosts in the Grid. ACRMU matches this information collected at IAU with the job's requirements and finds the most suitable host for that job. ACRMU passes this information to JSU, that does the task of job submission to the selected host.

Central to the design of AGMetS is the idea of an availability index vector for a host. The elements of this vector specify the failure probability of the resources. An Availability Provider program runs on any host that pools its resources in the grid. We assume without loss of generality that a method to periodically compute the availability index vector for a host is readily available. For our experiments, we make this method output random values as the availability index vector.

4. Prototype Implementation

To effectively make use of Grid resources, a basic Grid infrastructure is required that enables a Grid user to discover resources and run their applications on those resources. Further, a strong security mechanism should be in place that is able to perform authentication, communication protection, and authorization. The current prototype implementation of AGMetS assumes that this infrastructure is provided by Globus Toolkit 3 [1] (GT3).

4.1. IAU

This component is responsible for periodically gathering the relevant Service Data Elements (SDEs) from hosts. It does this by subscribing to Resource Information Provider Service (RIPS) of all the hosts in the Grid. IAU receives continuous notifications for desired SDEs from all the hosts, that are stored in a local cache (called Service Data Cache).

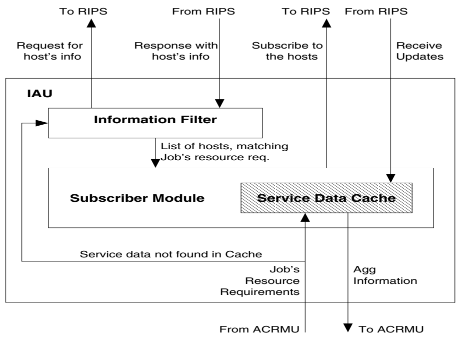

Figure 2. IAU Architecture.

The scheme, however, is not scalable when there are large numbers of hosts in the grid. In that case, the information collected at IAU would be enormous in content and therefore hard to manage. To overcome this problem we divide the job requirements into two parts- Resource requirements and QoS requirements. Resource requirements of a job refer to the platform needs of the job. It include OS type, processor type, processor speed etc. requested by the job. The QoS requirements of a job refer to its desired QoS needs, and include the required host availability, the maximum sustainable processor load etc. Now, instead of subscribing to all hosts in Grid, we first find the hosts that match the application's resource requirement, and then subscribe to only those hosts. This way the information that needs to be managed at IAU, is significantly reduced.

Figure 2 shows IAU architecture. Information Filter gets the job's resource requirements from ACRMU. It queries to RIPS of all hosts to get the relevant service data from all the hosts. Information Filter then selects the hosts that match the job's resource requirements, and sends this list of hosts to the Subscriber. Subscriber, in turn, subscribes to the RIPS of the selected hosts for the relevant service data. The returned updates are cached in Service Data Cache for ACRMU's lookup.

4.2. ACRMU

ACRMU fetches SDEs from IAU. It also receives the job information from UI. It then determines the suitable host to run the job on, based on the job requirements. It passes the name of this host along with the job specifications to the Job Submitter component. Figure 3 shows the detailed architecture of ACRMU. The main functional components in ACRMU are Job Requirement Parser, Resource Matchmaker, Scheduler and Monitor. These components are implemented as independent threads. There is a Job Status Table in this unit to keep track of the different jobs' status. For all the jobs the system and have not completed, there is an entry in the Job Status Table.

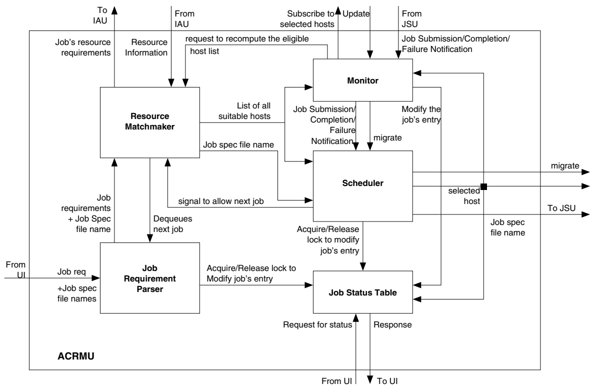
To IAU
From IAU
Subscribe to selected hosts
Update
From JSU
Job's resource requirements
Resource Information
request to recompute the eligible host list
Job Submission/Completion/ Failure Notification
Monitor
Resource Matchmaker
List of all suitable hosts
Job Submission/ Completion/ Failure Notification
Modify the job's entry
Job spec file name
migrate
Scheduler
migrate
selected host
To JSU
signal to allow next job
Job requirements + Job Spec file name
Dequeues next job
Acquire/Release lock to modify job's entry
Job spec file name
From UI
Job req
+Job spec file names
Job Requirement Parser
Acquire/Release lock to Modify job's entry
Job Status Table
Request for status
Response
ACRMU
From UI
To UI

Figure 3. ACRMU Architecture.

Job Requirement parser maintains a job queue for the requested jobs. A job that is submitted to AGMetS, is first added to this queue. It remains in the queue till AGMetS finishes servicing all the previously submitted jobs. A node in the job queue consists of the resource requirements and QoS requirements of the job, along with the job specification file name. Job Requirement Parser receives the job requirement XML file and the job specification XML file as the input. When it gets this new job request, it first tries to find a free entry in the Job Status Table. Once it finds a free entry, it obtains the lock on that entry of the Job status table, modifies it to reflect new job's status, and releases the lock. It then parses the job requirement file and adds an entry for the corresponding job in the job queue.

Resource Matchmaker dequeues one node from job queue after the previous job is submitted to the selected host. Resource Matchmaker then requests IAU for the service data by sending it the job's resource requirements. In response, it receives the relevant SDEs from IAU. It then parses SDEs' contents, taking the job's QoS requirements as the filter, and finally gives the list of all suitable hosts to Scheduler as well as to Monitor. The idea behind giving the entire suitable host list to the Monitor is that in case the need arises to migrate a particular job to other host, the new host can readily be selected from this list. The hosts in this list might not be suitable any more; in which case Resource Matchmaker is asked to recompute this list again.

Scheduler takes the list of suitable hosts as the input and outputs one host name randomly. The selected host name is sent to JSU (so that the job can be submitted to this host) as well as to the Monitor (so that monitoring of this host can start). Before sending the selected host name to JSU, Scheduler first acquires a lock on the respective entry of the Job status table, modifies the job's status to reflect that the job is now submitted to this host, and releases the lock. Jobs are serialized in the system, i.e. only when a job is submitted and the notification (of successful submission) has come, the next job will be served. If we allow a new job to proceed before getting the submission/failure notification from the monitor, then the correctness of the system might get compromised. (For example, a job is submitted to a host, and the monitor has not got the notification about successful job submission. Now, the information at the IAU for the corresponding host is inconsistent till the next information update comes from RIPS just after job submission, as job submission invariably changes some service data such as load, availability etc. A new job, which may come in this time interval, can get scheduled based on stale information).

Monitor subscribes to all the hosts in the host list (passed to it by Resource Matchmaker). It then continuously receives the availability updates from these hosts. Until Monitor gets job completion notification (from JSU), it continuously monitors the availability of the host where the job is submitted. If in this time interval, Monitor gets the low availability update for the concerned host, it proceeds towards migration of the job. For this, the monitor looks into the list of eligible hosts it maintains, and selects those who still qualify availability requirements of the job. If no host in this list is eligible anymore, then monitor asks the Resource Matchmaker to compute the host-list afresh. Monitor passes this list (along with the migration request) to the Scheduler. Scheduler selects one host from this list and sends this host name (along with the migration request) to JSU. JSU performs rest of the actions to ensure the migration of the job to the new host. JSU does this by sending the signal to the host to checkpoint the running job and send the checkpoint file back to the JSU. JSU then sends this checkpoint file to the new host. Another duty of monitor is to keep the Job Status Table up to date. For this, whenever the monitor gets job status notification from JSU, it locks the respective entry in the job status table, makes appropriate changes in the concerned job entry (i.e. changes the Job Status field) and releases the lock.

4.3. Job Submission Unit

Job Submission Unit, as shown in Figure 4, has a submitter component and a migration initiator component. Submitter gets the selected host name and specifications of the job from ACRMU, and submits the job to the selected host. Submitter also receives the completion/submission/failure of job message from the host where the job is running. Job Submitter simply forwards this message to ACRMU. Migration Initiator is responsible for migrating a currently running job for which the availability of the host (or memory/processor/RAM etc) has gone down. When it receives the migration request from Monitor, it asks the concerned host to checkpoint the job in question and send it back. It then sends this checkpoint file to the submitter to reassign it to the next selected host.

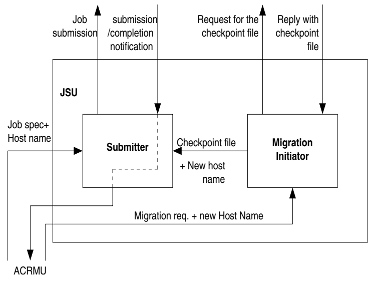
Job submission
submission /completion notification
Request for the checkpoint file
Reply with checkpoint file
JSU
Job spec+ Host name
Submitter
Checkpoint file
+ New host name
Migration Initiator
Migration req. + new Host Name
ACRMU

Figure 4. JSU Architecture.

5. Experimental Details

Our experimental testbed consists of four uniprocessor machines running Linux. The machines are chosen with varied configurations to emulate a heterogeneous Grid environment. All four machines are used for job execution, while only one machine hosts AGMetS prototype. The application used in this experiment performs a simple arithmetic operation in a finite loop.

5.1. Availability Improvement

To find the availability improvement with AGMetS, we run the application on our testbed, and simulate a host failure. For this, we force the availability index calculator of

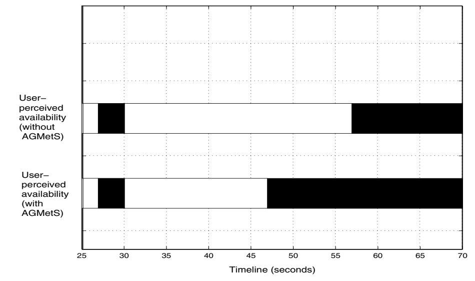

Figure 5. Availability improvement.

a host to output zero availability at some point in time after job starts execution on that host, and soon after crash that host. We repeat this process first when job is scheduled without the help of AGMetS and second when it is scheduled with the help of AGMetS.

Figure 5 shows user-perceived availability for both the cases. For each point *t* along the horizontal axis, a solid bar indicates that, at time *t*, the job is running. This means that at time *t*, there is an available Grid environment for job's execution. The first gap in the interval [25.0, 26.9] indicates that the job is submitted to a host but it has not started execution. The next gap (starting at time 30.0) indicates that the job is unable to run because the host has crashed. If the job runs as a part of some service, this gap simply means that the service becomes unavailable for this duration. In any case, this gap, therefore, means that the Grid environment has become unavailable for the job's execution. When job is scheduled without AGMetS, we optimistically assume that the human intervention (to restart the job on some other suitable host after the crash) occurs exactly when the crash occurs. We also assume that there is a mechanism to find the suitable host after the crash, and that mechanism is as good as that in AGMetS. When the job is scheduled with AGMetS, AGMetS detects the failure, and dynamically reschedules the job on some other host. Figure 5 shows the availability improvement in this case. Note that in the case where job is scheduled without AGMetS, results are taken under very optimistic assumptions. Relaxing these assumptions results in much larger gap, hence more relative availability improvement for the case where job is scheduled with AGMetS.

5.2. Job Response Time Improvement

We measure the job response time, first when job is scheduled without the help of AGMetS and second when it is scheduled with the help of AGMetS. To measure this, we run the same application on our testbed and again simulate a host failure.

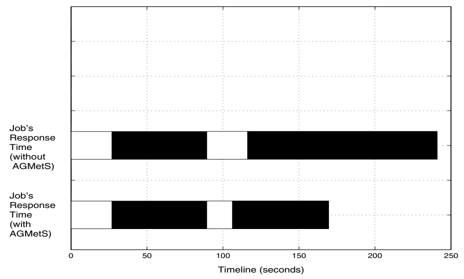

Figure 6. Job Response Time improvement.

Figure 6 compares the job response time for both cases. In the first case (when the job is submitted to a host without the help of AGMetS), the job starts execution at time 26.9 seconds. The host crash occurs at time 89.2 seconds. We restart the job on other host under the same optimistic assumptions about human intervention we make in the earlier experiment. The job completes execution at time 240.66 seconds. Next, we submit the job with the help of AGMetS, and repeat the above process. We assume that the application is built for checkpointing. To emulate the effect of migration, we record the number of iterations already made by the application, at the time of host crash. When AGMetS restarts the application on some other host, we run the application for remaining iterations only. This time, the job completes execution at time 169.31 seconds. The reason for this reduction in response time is timely detection of failures and subsequent job migration, by AGMetS. The computation already done when the failure occurs is not lost in this case. This shows the autonomous behaviour of AGMetS towards resource failures.

6. Conclusion and Future Work

In this paper, we have introduced the notion of scheduling based on "availability", and provided AGMetS framework for QoS adaptive Grid meta scheduling. We have also provided a prototype implementation of AGMetS. Experimental results show the gains in availability with our AGMetS prototype. Improvement in job response time as a result of AGMetS's autonomous behaviour towards resource failures is also shown. By combining resource brokering with fault tolerance, it proves to be a novel framework for Grid meta scheduling.

The experiments express the need of a good cache management scheme in the prototype. We plan to refine the prototype by implementing a more efficient cache management scheme in future. Preliminary tests have been conducted on a rather small Grid test bed. We plan to test the scalability of our prototype on a larger testbed. Also, AGMetS does not support a notion of advance resource reservation. We leave the extension of AGMetS to support advance resource reservation as future work.

References

[1] I. Foster, C. Kesselman, J. Nick, and S. Tuecke, "The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration", *Open Grid Service Infrastructure WG, Global Grid Forum*, 2002.

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

[3] F. Berman, R. Wolski, S. Figueira, J. Schopf, and G. Shao, "Application-Level Scheduling on Distributed Heterogeneous Networks", *Proceedings of Supercomputing*, 1996.

[4] D. Abramson, R. Buuya, and J. Giddy, "A Computational Economy for Grid Computing and its Implementation in the Nimrod-G Resource Broker", *Future Generation Computer Systems*,18(8), 2002.

[5] X. Zhang, D. Zagorodnov, M. Hiltunen, K. Marzullo, R. Schlichting. "Fault-tolerant Grid Services Using Primary-Backup: Feasibility and Performance", *Cluster 2004*, Sept. 20-23 2004, San Diego, California.

[6] I. Foster, C. Kesselman, C. Lee, R. Lindell, K. Nahrstedt, A. Roy. "A Distributed Resource Management Architecture that Supports Advance Reservations and Co-Allocation", *Intl Workshop on Quality of Service*,1999.