SEAMSTER: A Generic Fault Tolerant System for Dynamic Scheduling in a Distributed Real-Time System

Piyush R Satapathy
Piyush@cs.ucr.edu

Van Lepham
lephamv@cs.ucr.edu

Department of Computer Science and Engineering
University of California Riverside

Abstract:

For decades parallel and distributed computing has been a motive for time consuming tasks in-order to get the tasks done at a higher rate and to free the resources as early as possible. Parallel execution can be done in many ways ranging from a multi processor machine to a highly clustered grid. Some of the common resources used nowadays are clusters in an LSF (Load Sharing Facility), clusters in a Grid, or a set of machines in a LAN or WAN network or a single machine with number of processors embedded in it. We develop a framework called SEAMSTER which can execute parallel jobs in any of these resources irrespective of which version the tool is or which vendor the tool is provided from. Also we provide monitoring and feedback mechanisms for the executed jobs to make the whole scheduling a better performance, scalability, flexibility and extensibility. We present experimental results for several scheduling strategies that effectively utilize the monitoring and job-tracking information provided by SEAMSTER. These results demonstrate that the tool proposed can effectively schedule work across a large number of distributed processors or machines or clusters that are owned by multiple units in a virtual organization in a fault-tolerant way in spite of the highly dynamic nature of the resources used and complex policy issues. The novelty lies in use of effective monitoring of resources and job execution tracking in making scheduling decisions and fault-tolerance across a wide variety of resources. We have also visited some of the tools in the dynamic scheduling in distributed real-time system and we verify that our framework is more of a general version compared to all those.

1. Introduction:

For decades there has been a well adoption for parallel execution of time consuming tasks. Computer Scientists, Engineers and Software Industries nowadays use massive distributed computing to speed up the execution process. Tasks ranging from running regression test cases and running nightly software build in software industry to executing complex scientific calculations in research and development are all easily parallelized and executed in parallel. The user specifies the hierarchies of all the parallel jobs to be executed in order to get the proper final output. Depending on this user input the execution happens in parallel on a multi processor machine or across a number of interconnected machines. Distributed engines such as GRID and LSF are also providing such kind of parallel execution environments. But the problem lies if one or more jobs get failed due to some fault causing behaviors. Here we provide a generic system; an easy-to-use and application oriented tool which can run jobs in parallel in any parallel execution environment in the most fault tolerant way. The tool can run parallel jobs in any distributed engine such as any kind of Grid engines and LSF engines, or in a multiprocessor machine, or in a set of interconnected machines. Distributed engines such as Grid and LSF have their own scheduling policies by a spooling technique. But as we present a generic tool we implement the scheduling policies assuming that anytime we

can submit the number of jobs equal to the number of parallel execution possible in the engine. This saves the spooling overhead and gives a fair share with other users. Also we implement the scheduling for a set of interconnected machines and for a multiprocessor machine based on the well established and recently developed scheduling algorithms. We also present the experimental results of all these implemented algorithms with some well proved benchmarks.
In this paper, we present the architecture and characteristics of SEAMSTER and its performance on a set of platforms like a native machine, a set of machines and sets of different clustering environments. We consider a multi processor machine inside our campus for parallel execution in a single machine. For set of machines we select 40 machines inside the campus and for running in clusters we use GRID3 [3], a worldwide consortium of university resources consisting of 2000+ CPUs. These results show that SEAMSTER can effectively reschedule jobs if one or more of the machine stops responding due to system downtime or slow response time improve total execution time of an application using information available from monitoring systems as well its own monitoring of job completion times and manage policy constraints that limit the use of resources. These features demonstrate the effectiveness of SEAMSTER in overcoming the highly dynamic nature of the today's parallel job execution requirements and complex policy issues to utilizing the various available resources, which is an important requirement for executing large production jobs in an organization.

The rest of the paper is structured in the following manner. Section 2 discusses the state-of-the-art – the way that most recent tools handle the jobs in dynamic resource allocation environment and the difficulties faced. Section 3 discusses the characteristic behaviors of our proposed framework. Section 4 outlines our architecture in detail and component wise which addresses the scheduling woes described in section 2. Section 5 provides the implementation details of our tool and next section discusses the evaluation methods and experimental results which gives evidence of the effectiveness and comparison of different scheduling methodologies using the fully functional set of machines as the test bed. Future works and conclusion are discussed on Section 7.

2. Related Work:

We have explored a number of academic and industry oriented tools which are available widely. The most recent work on this is the SPHINX tool [11], which schedules the jobs across a number of grid sites available in Grid3 project. This handles fault tolerant behaviors; however, this work can not support to the other parallel execution environment. Similarly, the tool OpenSTARS (Open Schedulability Tool for Analysis of Real-time Systems) [1], is described as an "efficient, scalable, flexible, and extensible open-source tool and framework that real-time systems researchers and practitioners can use for both offline and online analysis" [1]. However, this tool does not provide sufficient scope for dynamic execution with fault causing behaviors. Besides these two tools, we also visited five other tools named as Cheddar, VEST, STAF, TimeWiz, RapidRMA, which give scopes either for a real time analysis of dynamic scheduling methods or for some specific application oriented systems, but are not so well designed to handle the fault causing behaviors. None of these tools efficiently presented a way to handle real life parallel job execution methods taking all possible fault causing behaviors into consideration. We visit all these tools briefly in the following section.

2.1 SPHINX

SPHINX [11] is a middleware scheduler in a grid system. It is a modular system including a client and a server. First, the user needs to pass the defined format execution request to the SPHINX client. The client will forward the job to the server. The server acts as a scheduler. It will locate the best resource to schedule the job. It also maintains catalogs of the data and replicas; estimates the completion time of the requests; as well as monitors the status of the resources. After receiving the decision from the server, the client will submit the job request to the grid system. The SPHINX handles the fault tolerant behaviors using the Job Tracker module lies within the client. The Job Tracker will keep track the status of the submitted jobs. If some fault causing behavior happens, it will send the cancellation request to the grid to cancel the executing job and send re-schedule request to the server to request for a replacement resource.

However the architecture is not robust enough to be fit into any parallel execution environment. The whole purpose of this architecture was to model a fault tolerant system for scheduling jobs across a number of grid sites. In this paper, the scheduling algorithms and the architecture have been designed keeping the grid site as the granularity rather than a single machine in the grid cluster. The later is important when there are hundreds of machines connected to a particular grid cluster.

2.2 OpenSTARS

The authors of OpenSTARS [1] tool pay much attention on some required criteria of the tool to be considered as a good real-time analysis tool. These criteria include correctness, performance, scalability, flexibility, and extensibility. The OpenSTARS operates based on these main objects: Domain, SchedAlgorithm, Driver, and Result. The Domain object is an interface used to translate from and to the format recognized by the scheduling algorithm. It is called Task, Resource, and Dependency model (TRDModel). The SchedAlgorithm object will do the job of analyze the system and report if and how quickly the algorithm return a correct result. The Driver object acts as an interface between real-time system and scheduling algorithm. It will choose the best algorithm. The Result object keeps the scheduling analysis result and domain information for usage when needed. The operation includes three steps: setup, analyze, and interpret. In the setup step, the Domain will parse the XML input and translate to the TRDModel. In the second step, the Driver will scan the TRDModel to collect information about the jobs and resources. Based on this information, the Driver will choose a correct algorithm to run and call the corresponding SchedAlgorithm. Then the SchedAlgorithm will return a Result object. In the last step, the Domain will save and output the result to the user in the format of an XML file.

OpenSTARS does not provide a fault tolerant technique of scheduling jobs. It is more or less an experimental tool rather than a real life system for distributed job scheduling.

2.3 Cheddar

Cheddar [13] is an open-source real-time scheduling tool. It implements most of the classical real time scheduling algorithms such as Rate Monotonic Analysis, Earliest Deadline First, Deadline Monotonic, and Least Laxity First.. However, it does not support the fault tolerant behaviors and can not be compatible with wide variety of resources.

2.4 VEST

VEST [14] has been built as a toolkit whose aim is to provide a rich set of dependency checks based on the concept of aspects to support distributed embedded system development via components. Though this tool is a real time distributed system, yet it can not be applied across applications. It is limited to embedded applications and lacking the proper ways to handle the fault tolerant techniques.

2.5 STAF

The Software Testing Automation Framework (STAF) [16] built by IBM is an execution engine which helps to automate the distribution, execution and analysis of results of the test cases in an environment properly arranged for software nightly built. This is an open source, multi-platform, multi-language framework designed around the idea of reusable components, called services (such as process invocation, resource management, logging, and monitoring). It is built on top of XML, python and also provides a nice GUI monitoring application. However this can not schedule any real life jobs in a parallel environment. Also STAF does not take care of rescheduling the failed jobs as its application is not mattered with the handlings of the fault causing behavior.

2.6 TimeWiz

TimeWiz [12] is another tool designed for modeling, analyzing, and simulating performance and timing behavior of real-time systems. TimeWiz only works in the Widows platform and it is more an analysis tool rather than a real time distributed system.

2.7 RapidRMA

RAPID RMA [15] is also a real time analysis tool which allows real-time systems software developers to prevent costly design mistakes and accelerate their development schedules. The multiple analysis tools contained in RAPID RMA allow designers to test software models against various design scenarios and evaluate how different implementations might optimize the performance of their systems. By isolating and identifying potential scheduling bottlenecks in both soft and hard real-time systems, RAPID RMA gives a better hand to the art of modeling in real-time system.

3. Characteristics of SEAMSTER:

Our main goal is to put forward an infrastructure which will execute parallel jobs in a fault-tolerant scheduling technique. Fault behaviors in a real time execution system are caused by various reasons such as geographically distributed systems, heterogeneous resources, dynamic loads, availability of the resources, decentralized ownerships, and different local scheduling policies. We see the consequences as below for these fault behaviors. (1) Job Failure with No Output: job will be stopped running without completion. (2) Job Failure with Wrong Output: job will complete running but it will produce wrong output due to some inter dependencies or some synchronization issues. (3) Job Slowdown: job will be continuing running at a very slow rate to exceed the time out period. To overcome this problem we implement the below three techniques.

1. Feedback Technique: We develop a method to get the execution status information of each submitted job. Our scheduling algorithm utilizes this information to submit any new job, kill and reschedule any slow job, and reschedule the failed jobs.

2. Monitoring Technique: We develop a method to monitor all the machines on the parallel execution environment. This information is updated frequently and our scheduling algorithm uses this information for choosing the right resource. We reschedule jobs if one or more of the jobs exceed time out period by system downtime or slow response. In the implementation across machines we utilize the monitoring information not to send the jobs to the failed or slowed down systems. We keep on looking to those systems until the systems pass some pinged criteria. We also schedule keeping the view of minimum number of resource used and minimum amount of time used.

3. Keeping History: We store the history of executed jobs for future reference. The framework keeps the records of the size and type of jobs executed for the first time in a database. Usually the user selects an algorithm to schedule the jobs dynamically. If the user doesn't specify a particular algorithm to run, then the analyzer chooses an optimized algorithm matching the information of the jobs to be scheduled with the information stored in the database. We also keep the trivial information like individual machine performance and algorithm related information into the database.

4. SEAMSTER Architecture:

Figure 1 describes the architecture of SEAMSTER. Targeting the three characteristics as mentioned above we develop 3 layers in our proposed architecture. The end levels are the user who schedules the job and the resource that executes the job. And in the middle layer we have three components named as analyzer, monitor and history storage. The monitor has 3 components such as a job monitor, a resource monitor and a supervisor. The details of each component have been given below.

4.1 Scheduling Job

The user schedules the job by giving input information to the system. Input information are the type of resource, the resource handling information, the static or dynamic job list and the choice of scheduling algorithm.

4.2 Analyzer:

The main function of the analyzer is to choose a best algorithm based on the type of jobs at hand. If the user chooses optimized algorithm as an option then the analyzer analyzes the information in the history storage.

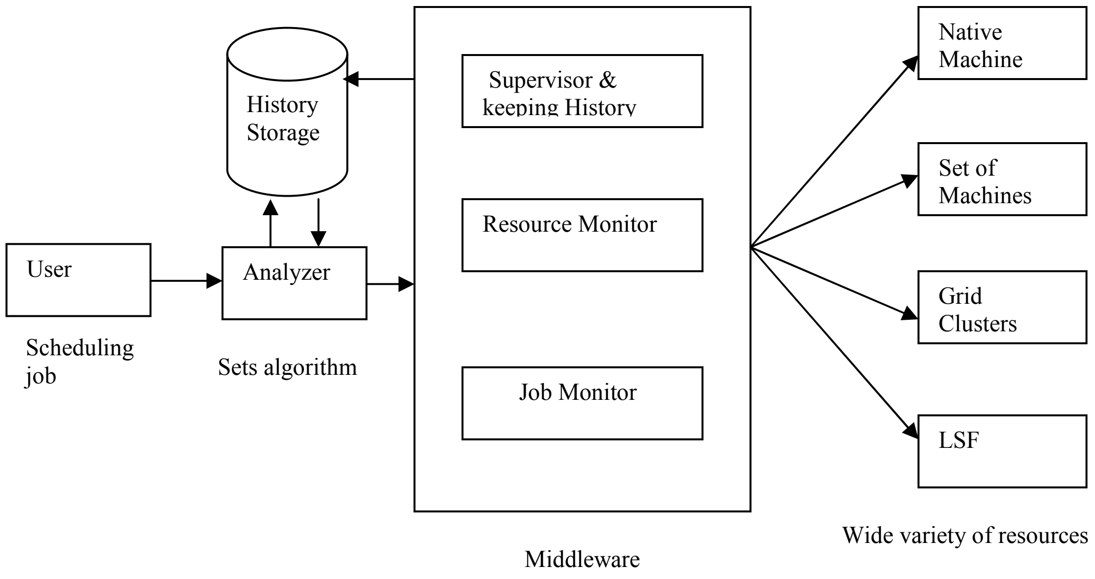


Figure 1: SEAMSTER system architecture

It matches the type of file and size of file of the jobs at hand with similar jobs information from the data base and then based on this information it allocates the best algorithm.

4.3 History Storage:

The history storage stores all the first hand information. It keeps a record of all the unique kind of jobs executed along with the time taken and the resource name used for the job. It also stores the information of all the faulty machines and jobs which got failed and need to be rescheduled.

4.4 Supervisor:

To avoid using RPC (Remote Procedure Calls), we design a module called Supervior. Supervisor is an executable program. It acts as a middleware helping the scheduler in starting, suspending, and resuming a job. It has three basic functions as followed:

1. Start: when the Scheduler asks the Supervisor to execute a job, the Supervisor will create a process to start executing that job. In the mean time it will collect the corresponding process ID and write to a file. The Scheduler will collect the process ID from that file and store in the database to use later to suspend or resume the job.
2. Suspend: when the Scheduler asks the Supervisor to suspend a job, it needs to provide the Supervisor the process ID of the required suspend job. The Supervisor will suspend that job by sending a SIGSTOP signal.
3. Resume: similarly, when the Scheduler asks the Supervisor to resume a job, it will need to provide the Supervisor the corresponding process ID of that job. The Supervisor will resume the job by sending a SIGCONT signal.

4.5 Resource Monitor:

Resource monitor monitors all the machine where jobs are scheduled. It keeps on pinging all the used resources. If any machine does not respond within a fraction of time second then the monitor assumes it as faulty and updated the history storage. All the faulty machines will not be monitored any more until the amount of jobs at hand get finished execution.

4.6 Job Monitor

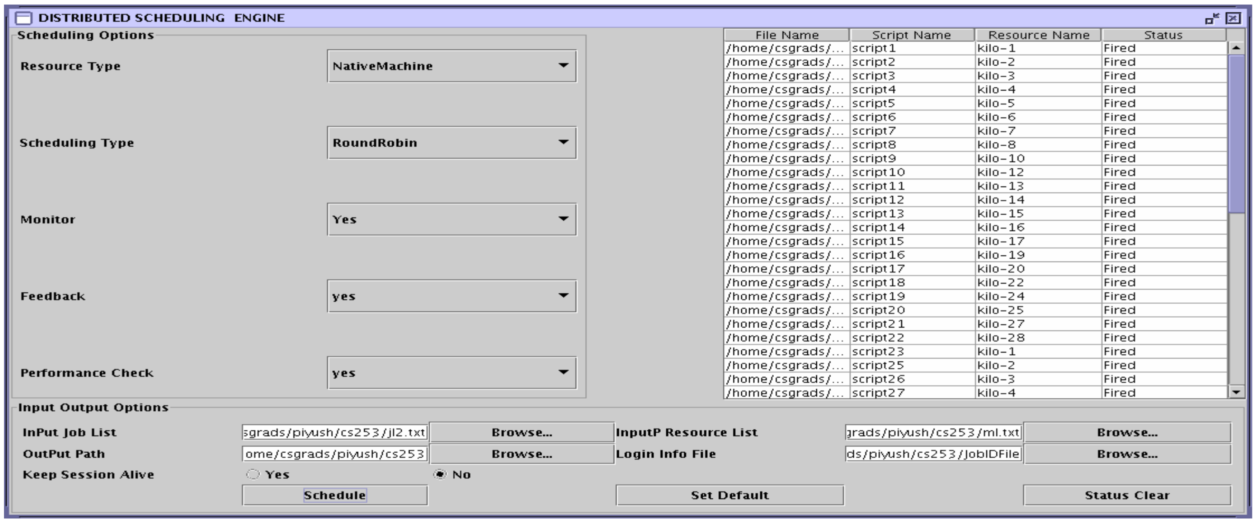
This component of the architecture keeps on looking to the job status of each job. We develop a method of knowing the status of job at three various stages such as fired, running and done. We keep a time track of every job scheduled and if the time spent for any interval of these three exceeds a certain specific amount we kill the job and reschedule it according to the algorithm planned before.

4.7 Wide Variety of Resources:

This is one of our major implementation. We have generalized the kind of resource a user wants to use. The use can give a set of machine names or a single machine or a cluster like grid or LSF or anything else. With addition to the type of resource the user needs to supply with the login information as a simple file format. The file should contain the command to loginto the resource, command to execute the job, command to know the status of the job and command to kill the job. For example if the user chooses grid as his resource then he must give “qsub” as execution command, “qstat” as status command and “qdel” as delete command inside a file in a certain prescribed format.

5. Implementations:

We have developed a user friendly Graphical User Interface in java aiming to ease the use of the system. Our main implementation of the system goes in C++ language and some scripting in PERL. The different components shown in the GUI in fig 2 have been described below.



Resource Type: NativeMachine
Scheduling Type: RoundRobin
Monitor: Yes
Feedback: yes
Performance Check: yes

File Name	Script Name	Resource Name	Status
/home/csgrads/...	script1	kilo-1	Fired
/home/csgrads/...	script2	kilo-2	Fired
/home/csgrads/...	script3	kilo-3	Fired
/home/csgrads/...	script4	kilo-4	Fired
/home/csgrads/...	script5	kilo-5	Fired
/home/csgrads/...	script6	kilo-6	Fired
/home/csgrads/...	script7	kilo-7	Fired
/home/csgrads/...	script8	kilo-8	Fired
/home/csgrads/...	script9	kilo-10	Fired
/home/csgrads/...	script10	kilo-12	Fired
/home/csgrads/...	script11	kilo-13	Fired
/home/csgrads/...	script12	kilo-14	Fired
/home/csgrads/...	script13	kilo-15	Fired
/home/csgrads/...	script14	kilo-16	Fired
/home/csgrads/...	script15	kilo-17	Fired
/home/csgrads/...	script16	kilo-19	Fired
/home/csgrads/...	script17	kilo-20	Fired
/home/csgrads/...	script18	kilo-22	Fired
/home/csgrads/...	script19	kilo-24	Fired
/home/csgrads/...	script20	kilo-25	Fired
/home/csgrads/...	script21	kilo-27	Fired
/home/csgrads/...	script22	kilo-28	Fired
/home/csgrads/...	script23	kilo-1	Fired
/home/csgrads/...	script25	kilo-2	Fired
/home/csgrads/...	script26	kilo-3	Fired
/home/csgrads/...	script27	kilo-4	Fired

InPut Job List: sgrads/piyush/cs253/jl2.txt
OutPut Path: ome/csgrads/piyush/cs253
Keep Session Alive: Yes [✓ No]
InputP Resource List: grads/piyush/cs253/ml.txt
Login Info File: ds/piyush/cs253/JobIDFile

Figure 2: Snapshot of SEAMSTER GUI

5.1 Scheduling Options

The system allows the user to choose an algorithm in the supplied list: Round Robin, CPU Based, Job Completion Based, Earliest Deadline First , Least Laxity First and etc. If the user wants the analyzer to calculate the best fitting algorithm the he has to choose the option of “optimized algorithm”. The system also allows the user to choose between a list of available resources that they want to use such as a Native Machine, a Set of Machines, GRID, or LSF or any other clustering environments. There are options for running the system without any monitoring and feed back mechanism. And the last option here we have is a performance check option; upon setting this option to yes the system will run the algorithms with some given bench mark to measure the performance.

5.2 Input Output Options

The user needs to supply a list of executable jobs that are stored in a file. The system allows the user to browse to the location where the input job list file is saved. Similarly the resource name is also set in a file and given as input. And the login information and the path to output are also given to the system as input. Each job in the job file is represented as a script name. The script contains the job to be run. Right now we don’t support how to create these scripts. But we have plans to make our systems robust enough to make these scripts automatically without depending upon the user to create it. We have an option named as “keep session alive”. If the user chooses yes then the system will behave dynamically and will allow real time jobs to be handled until the user closes or stops the application. Upon choosing this option as “no” the system will behave as a static scheduler.

5.4 Status Updates:

The status panel of the GUI shows a tabular format of the output in four different columns. The columns present the name of the job file, name of the job and name of the machine and status. The status column will change from time to time starting from fired to running and then done.

6. Evaluations and Setups:

We currently implement 5 different algorithms in our system. We have visited these algorithms shortly as shown below. Our experiment is based on a comparison of performance of all these algorithms with and without of monitor and feedback mechanism. Our machine set up is discussed in the second section

6.1 Algorithms Implemented

6.1.1 Round Robin:

Round Robin algorithm tries to submit jobs in the order of machines given in the machine list. Without waiting for the jobs to be finished in any one machine the scheduler schedules in a round robin fashion. The maximum limit of any machine is not exceeded. With the monitoring and feedback mechanism, the scheduler schedules on the machines which are active and promptly running.

6.1.2 CPU Based:

This algorithm takes the number of CPUs in each machine into account and schedules the job based on the maximum number of jobs that can be scheduled in any one machine. With the monitor and feedback mechanism the scheduler utilizes the information of previously submitted jobs from the history storage and calculates the load rate with the following formula for each machine it plans jobs to schedule

$\text{Rate}_{\text{machine}} = (\text{panned\_job}_{\text{machine}} + \text{unfininshed\_job}_{\text{machine}}) / \text{CPU}_{\text{machine}}$

Where “machine” represents a particular machine name.

6.1.3 First Come First Service based :

The system will schedule the job as long as there is an available resource. When all the resources are busy, the system keeps checking and waiting for the first job to be done and the corresponding resource to be released. Then it will schedule the next job to that resource.

6.1.4 Early Deadline First (EDF):

The system will collect the jobs in the input job list and build a priority queue based on the deadlines of the jobs with the smallest deadline being at the top. While there is job in the queue, the system will schedule the job that has smallest (or earliest) deadline first accordingly. When a new job arrives, it will update the queue. It also can suspend the running job and schedule the new job if the new job has smaller deadline, then resume the suspended job later when there is no competition.

6.1.5 Least Laxity First (LLS):

LLS algorithm will assigns the priority of the jobs based on their laxity values. The laxity value is calculated as followed:

$\text{Laxity} = \text{Time\_to\_deadline} - \text{Remain\_exec\_time}$

LLS will update the laxity values of all the jobs whenever there is a new job arrival and schedule the job with the least laxity first.

6.2 Set Ups:

We set 40 Machines inside the Engineering Building Unit II. Some of them permanently show fault causing behaviors; 5 Machines do not get ported, other 6 Machines get connected but fail to execute the jobs. We wrote 15 simple loop functions as part of our evaluation benchmarks. The execution time of these loop functions varied from 5 seconds to 5 minutes. We take each such loop function as one of our job. We have 60 such Jobs. Each job name is put in side a script and all the scripts are listed in a file line wise and this file is one of the inputs to our system. The different characteristics and results of these experiments are shown below.

7. Experiments and Results:

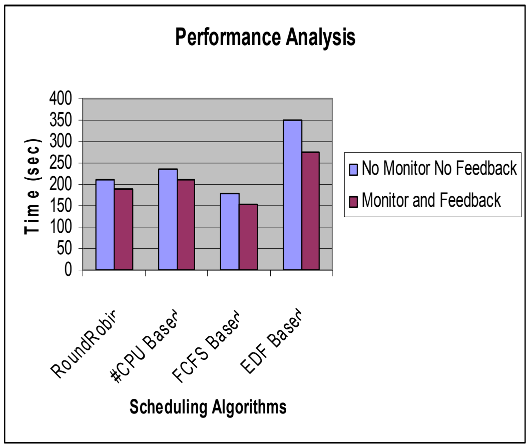


Figure 3a. Performance analysis on a list of machines

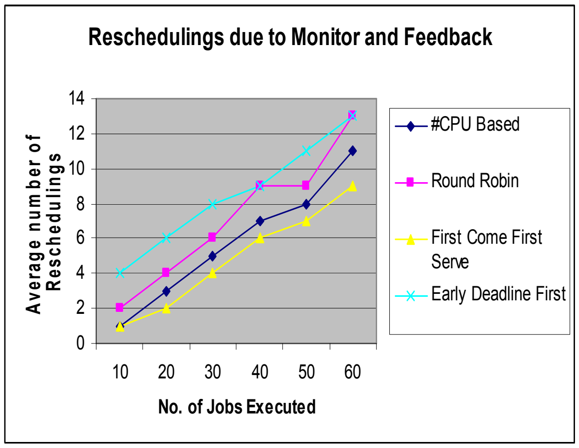


Figure 3b. Rescheduling on a list of machine as shown in fig 3a.

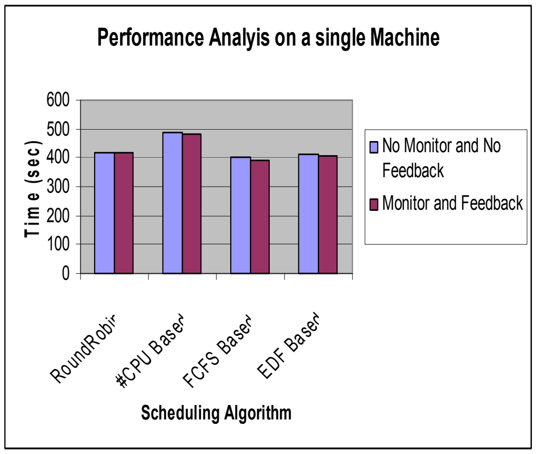


Figure 3c. Performance analysis on a Single machine

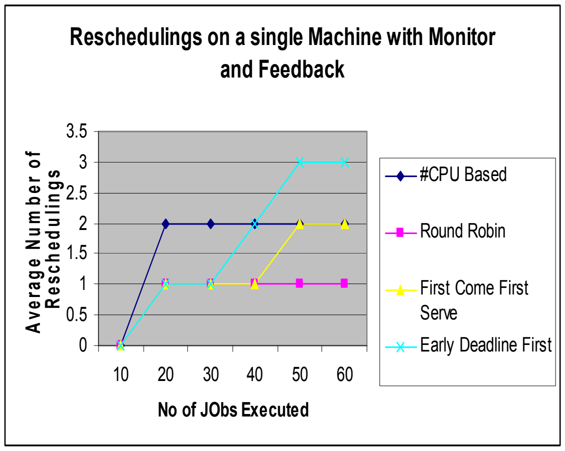


Figure 3d. Rescheduling on a single machine as shown in fig 3d.

We have plotted the results of two experiments. The first experiment was on a set of 40 machines with the set of 60 jobs to execute and the second experiment was on a single machine with 60 jobs to execute in parallel. From fig. 3a we observe that monitoring and feedback mechanism gives a better performance up to 10%. The time note here is the end to end execution time. With no monitoring and feed backing, the system will hang out and give no outputs of the jobs executed in the faulty machine. The time we noted for this is the time out of the system itself. With monitoring we kill the jobs on faulty machines with in a fraction of recognizing the faults and reschedule them in good available machines on the basis of the algorithm chosen.

Fig. 3b shows the number of rescheduling in each of these algorithms. We do the experiments several times and take the average number for plotting. Similarly fig. 3c and fig. 3d show the time consumed and rescheduling done in one particular machine. However the rescheduling is very less here because of the high power 2.8GHz dual Intel Xeon processor machine. But there are certain rescheduling due to the time out for certain jobs. And this is because the machine is not capable of running 60 jobs in parallel and it automatically does a serialization of jobs. The time consumed by all these algorithms is also significantly higher than that of the first experiment.

8. Conclusion and Future Work:

In this paper we introduced a new architecture for parallel execution of jobs in real life. We visited a number of tools across industry and as well as academics and we figured out that there was no such robust tool which could fit into a dynamic environment of today's first changing software development and methodologies. We took into account a various number of resources and the best fault tolerant techniques that has been devised yet and all the well established dynamic scheduling algorithms. Our initial . experiments on a certain number of machines show a promising result. We have not yet tried our system in cluster environment.

We are planning to get a planet lab ID and an ID from Grid3 to conduct more research and experiments on our proposed system. Though we initially targeted for a number of algorithms we could not finish all of them due to the time constraint and also our planning regarding the implementation of certain benchmarks into our system are yet to be done

References:

[1] K. Bryan, T. Ren, J. Zhang, L. Dipippo, and V. Fay-Wolfe. The Design of the OpenSTARS Adaptive Analyzer for Real-Time Distributed Systems. In Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium IPDPS'05, 2005.

[2] C. Cavanaugh and R. Ari. Dynamic Resource Management Algorithm for a Distributed Real-time System. In Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium IPDPS'05, 2005.

[3] I. Foster. The Grid2003 Production Grid: Principles and Practice. In Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing, 2004.

[4] N. I. Kaemeno_ and N. H. Weiderman. Hartstone distributed benchmark: requirements and definitions. In Proceedings of the 12th IEEE Real-Time Systems Symposium, IEEE Computer Society Press, December1991.

[5] R. Kar and K. Porter. Rhealstone- a Real-Time benchmarking Proposal. In Dr Dobbs Journal, (2), February 2002.

[6] D. L. Kiskis and K. G. Shin. SWSL: A Synthetic Workload Specification Language for Real-Time Systems. In IEEE Transaction on Software Engineering 20(10), October 1994.

[7] K. Low, S. Acharya, M. Allen, E. Faught, D. Haenni, and C. Kalbeisch. Overview of Real-Time Kernels at the Superconducting Super Collider Laboratory. In Proceedings of the IEEE Particle Accelerator Conference, 1991.

[8] B. Shirazi, L. Welch, B. Ravindran, C. Cavanaugh, B. Yanamula, R. Brucks, and E. huh. DynBench: A Dynamic Benchmark Suite for Distributed Real-Time Systems. In Proceedings of the 11 IPPS/SPDP'99 Workshops Held in Conjunction with the 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing,1999.

[9] N. D. Thai. Real-Time Scheduling in Distributed Systems. In Proceedings of the International Conference on Parallel Computing in Electrical Engineering PARELEC'02, 2002.

[10] B. G. Ujvary and N. I. Kamenoff. Implementation of the Hartstone distributed benchmark for real-time distributed systems: results and conclusions. In Proceedings of the 5th International Workshop on Parallel and Distributed Real-Time Systems, IEEE Computer Society Press, 1997.

[11] J. uk In, P. Avery, R. Cavanaugh, L. Chitnis, and M. Kulkarni. SPHINX: A Fault-Tolerant System for Scheduling in Dynamic Grid Environments. In Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium IPDPS'05, 2005.

[12] TimeSys Corporation, TimeWiz: Model and Analyze System Performance http://www.timesys.com/products/timewiz/

[13] The Cheddar Project: a Free Real Time Scheduling Analyzer http://beru.univ-brest.fr/~singhoff/cheddar/

[14] J.AStankovic, R. Zhu, R. Poornalingram, C. Lu, Z. Yu, M. Humphrey, and B. Ellis, "VEST: An Aspect-Based Composition Tool for Real-Time Systems," The 9th IEEE Real-Time and Embebed Technology and Applications Symposium, Toronto, Canada, May 2003.

[15] RapidRMA: The Art of Modeling Real-Time Systems
http://www.tripac.com/html/prod-fact-rrm.html

[16] Software Testing Automation Framework (STAF)
http://staf.sourceforge.net/index.php