Job-Site Level Fault Tolerance for Cluster and Grid environments*

Kshitij Limaye[1], Box Leangsuksun[1], Zeno Greenwood[1], Stephen L. Scott[2], Christian Engelmann[2,3], Richard Libby[4] and Kasidit Chanchio[5]

[1]Louisiana Tech University, Ruston, LA 71270, USA
[2]Oak Ridge National Laboratory, Oak Ridge, TN 37831, USA
[3] The University of Reading, Reading, RG6 6AH, UK
[4]Enterprise Platforms Group, Intel Corporation
[5]Thammasat University, Thailand

ksl007@latech.ed, box@latech.edu, greenw@phys.latech.edu, scottsl@ornl.gov, engelmannc@ornl.gov, richard.m.libby@intel.com, kasidit@cs.tu.ac.th

ABSTRACT

In order to adopt high performance clusters and grid computing for mission critical applications, fault tolerance is a necessity. Common fault tolerance techniques in distributed systems are normally achieved with checkpoint-recovery and job replication on alternative resources, in cases of a system outage. The first approach depends on the system's MTTR while the latter approach depends on the availability of alternative sites to run replicas. There is a need for complementing these approaches by proactively handling failures at a job-site level, ensuring the system high availability with no loss of user submitted jobs. This paper discusses a novel fault tolerance technique that enables the job-site recovery in Beowulf cluster-based grid environments, whereas existing techniques give up a failed system by seeking alternative resources. Our results suggest sizable aggregate performance improvement during an implementation of our method in Globus-enabled HA-OSCAR. The technique called "Smart Failover" provides a transparent and graceful recovery mechanism that saves job states in a local job-manager queue and transfers those states to the backup server periodically, and in critical system events. Thus whenever a failover occurs, the backup server is able to restart the jobs from their last saved state.*

* Research supported by Department of Energy contract DE-FG02-05ER25659 and Center for Entrepreneurship and Information Technology, Louisiana Tech University..
+Research supported by the Mathematics, Information and Computational Sciences Office, Office of Advanced Scientific Computing Research, Office of Science, U. S. Department of Energy, under contract No.DE-AC05-00OR22725 with UT-Battelle, LLC.

1. Introduction

Grid computing [1] is fast becoming a promising technology due to the collaboration opportunities it creates for organizations to work together to achieve common goals through resource sharing. As more and more critical applications shift to the Grid platform, it becomes increasingly important to ensure their high availability and fault tolerance.

Collaborating organizations usually provide individual high performance clusters as resources which contribute towards the computational power of the grid. Though the nature of the Grid is distributed, inevitable failures can make a *site* (a member of the Virtual Organization (VO), which can be a computational cluster resource of that VO) unusable, reducing the number of resources available and in turn, slowing down the overall speed of computation [2].

A cluster (Beowulf style) head node mostly acts as a single entry point to a site and provides necessary services, such as job schedulers. If job sites are made up of clusters, then the failure of the single head node of a cluster causes these services to be unavailable for the time that the head node is nonfunctional. HA-OSCAR [3] removes this single point of failure using component redundancy and imparts self healing capabilities for critical HPC services. While some approaches [13][14][17] leave a failed job site to heal on its own, we focus on guaranteeing the high availability of the site coupled with job-level fault resilience.

The current active/hot-standby model in HA-OSCAR provides an excellent solution for stateless services, where the transition from the primary head node to the backup is executed smoothly. However, this mechanism is not

graceful if stateful services, such as job management, are involved. Our research effort concentrates on such a graceful transition to the backup. We aim to provide a transparent recovery that includes a completion of currently running jobs after a failover.

This paper proposes a Grid-aware fault resilient mechanism in a Beowulf style cluster based job-site and experimental study of the "Smart Failover" feature in HA-OSCAR. We aim towards a graceful recovery in terms of job management by monitoring the job queue and keeping the standby server updated with the changes to it..

When jobs are submitted through Globus [4], a job-manager is invoked to submit the job to the local scheduler and return the output to the submitter. However, in a case of job-site failure, this submission information is normally unrecoverable even if the job has completed but not yet returned. An enhancement of the Globus job submission mechanism is needed in order to be aware of the failure and be able to recover transparently. We have implemented the "Smart Failover" feature to address these issues.

This paper is organized as follows. Section 2 describes related and ongoing research. In Section 3, we present the proposed framework for Smart Failover in HA-OSCAR. Section 4 explains our algorithm used in updating the backup server and the necessary enhancement for Grid job retrieval clients. Implementation details and experimental setup are presented in Section 5. Section 6 discusses our results. Section 7 summarizes the presented research and a brief description of future work.

2. Related Research

Globus has become the de facto standard for grid computing. The Globus tool kit consists of a set of tools and libraries to support grid applications. Fault tolerance approaches in grid systems are commonly achieved with checkpoint-recovery and job replication [13][14] [17], which create replicas of running jobs and hoping that at least one of them succeeds in completing the job. Weissman and Womack [13] introduced a scheduling technique for a distributed system which suffers from increased job delays due to insufficient number of remote sites to run the replicas. Abawajy [14] achieved grid fault tolerance by scheduling jobs in spite of insufficient replicas. His approach requires at least one site to volunteer for running the replica before the execution can start. In [17], jobs replicas are submitted to different sites which return the checksum of the result. The checksums received from various sites are then compared to ensure whether majority results are the same, in order to avoid a result from a malicious resource, which delays the retrieval of result until a majority is reached. Therefore, job delay increase may result not only from failures but also from the verification overhead. Wrzesinska *et al* propose a solution [18] that avoids the unneeded replication and restarting of jobs by maintaining a global result table and allowing orphaned jobs to report to their grand-parent incase their parent dies. However, their approach is strictly for divide-and-conquer type of applications and cannot be extended to environments where the sub-processes require communication.

"Grid Workflow" [8] leaves recovery decisions to the submitter of the job via user-defined exception handling. Grid Workflow employs various task-level error-handling mechanisms, such as retrying on the same site, running from the last checkpoint, and replicating to other sites, as well as masking workflow level failure. Nonetheless, most task-level fault tolerant techniques,[8][13][14] attempt to restart the job on alternative resources in the Grid in an event of a host crash. Hence, there is a need for complementing these approaches by improving failure-handling at the site level, especially in a cluster computing environment.

LinuxHA [5] is a tool for building high availability Linux clusters using data replication as the primary technology. However, LinuxHA only provides a heartbeat and failover mechanism for a flat-structure cluster which does not easily support the Beowulf architecture commonly used by most job sites.

OSCAR is a software stack for deploying and managing Beowulf clusters [6][7]. This toolkit includes a GUI that simplifies cluster installation and management. Unfortunately, a detrimental factor of the Beowulf architecture is the single point of failure (SPoF). A cluster can go down completely with the failure of the single head node. Hence, there is a need to improve the high-availability (HA) aspect of the cluster design. The recently released HA-OSCAR software stack is an effort that makes inroads here. HA-OSCAR deals with availability and fault issues at the master node with multi-head failover architecture and service level fault tolerance mechanisms.

PBS [9] and Condor [16] are resource management software widely used in the cluster

community. While a HA solution [12] for Condor job-manager exists, there a dearth of such solutions for the PBS job manager. The failure of Condor Central Manager (CM) leads to an inability to match new jobs and respond to queries regarding job status and usage statistics. Condor attempts to eliminate the single point of failure (i.e. the Condor CM) by having multiple CMs and a high availability daemon (HAD) which monitors them and ensures one of them is active at all times. Similarly, HA-OSCAR's self-healing core monitors the `pbs_server` among other critical grid services (e.g. xinetd, gatekeeper etc), to guarantee the high availability in an event of any failure.

3. Proposed Framework

Consider a job site where 100 jobs have been submitted to a local cluster scheduler, a failure at the site-manager (e.g. a cluster head node) will result in the outage of the total site. Users either wait for the site-manager to be fully recover or they can use a HA solution such as HA-OSCAR to recover from the failure within seconds.

The current HA-OSCAR self-healing mechanism transparently provides HA for critical services at the site-manager. However, it does not support graceful service migration for job schedulers and thus users must resubmit the incomplete jobs to the standby head node. R. Rabbat and T. McNeal [20] give an example of NFS service migration, achieved by use of either shared storage between the primary and the standby to store critical files or by migration of these critical files to the standby. Similarly we need to update the standby (backup) with the temporary job files (containing a specification of the executable to use, `stdout`/`stderr`, status, etc) of submitted jobs to enable it to start those jobs gracefully on the backup. With the use of the above technique, in case of a head node outage, running and queued jobs will be automatically recovered on the backup.

Whenever there is a remote site failure, the connected clients need to either wait for the remote site to be available again or they need to connect to another alternative resource to get required service. G. Ahrens [15] describes the need for the client to be "Cluster Aware", i.e. being intelligent enough to connect to an alternative node in case of failure. In order to fully utilize the "Smart failover feature the naïve grid job clients need to be modified. The grid based job clients need to be "failover aware" to take advantage of the Smart failover feature. We address the issues by which grid job retrieval mechanisms, such as `globus-job-run`, `globus-job-submit` and `globusrun`, are unable to fetch the output/error log for submitted jobs after a remote site fails and then recovers, as the job handle/jobIDs become invalid after a failure. Hence there is a need to enhance the Globus job retrieval mechanism, so that a failover in the cluster job-site is transparent to the client. This removes the inability of the client to retrieve the job status and output/error log in case of remote failures. In addition, it alleviates the need for job replication to other sites, and hence preserves all computing-node resources when the outage is only caused by the head node.

3.1 Basic components

Figure 1 illustrates the HA-OSCAR smart failover mechanism in a Grid environment. The framework consists of 3 components: the *event monitor*, *job monitor and the backup updater*. Critical system events, such as repeated service failure, memory leaks and system overload, are analyzed by the *event monitor* using the HA-OSCAR monitoring core. The second component, *job monitor*, is a daemon that periodically monitors a job queue at a user specified interval. It can also be triggered by the *event monitor* in a case of critical events.

Whenever the job queue monitor senses a change in the job queues, it invokes the *backup updater* to synchronize the standby server with the changes in job queue and other critical directories. This approach, a combination of periodic and event triggered updates, helps to keep the standby server up-to-date with the current job queue status and results in a graceful failover procedure.

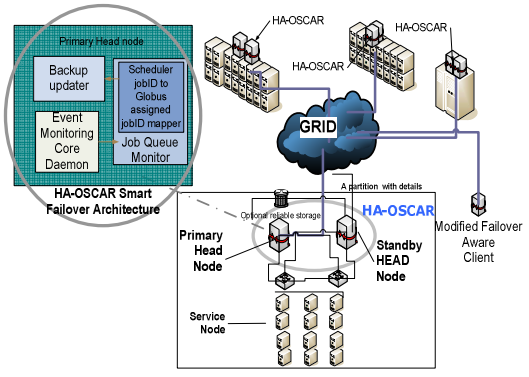

Figure 1 HA-OSCAR Smart Failover Feature in a Grid Scenario

3.2 Grid Aware Failover

A mapping between the Globus assigned job id (GjobID) and the scheduler assigned job id (SjobID) is the key information for transparent head node fail-over and job restart mechanisms in our HA-OSCAR cluster. Whenever a new job has been added, the job queue monitor determines whether it has been submitted through a Globus mechanism or the local scheduler job submission primitives. If the newly added job has been submitted through Globus, the job queue monitor maps the GjobID to SjobID. For a later retrieval by the client, this mapping is synchronized to the Globus job directory on the Standby server using the backup updater.

Considering a situation, where the site manager has failed (after an update) and the standby server has taken over: The failover-aware client is able to use the mapping from the `GjobID` to `SjobID` to find the correct status of his/her jobs.

The mapping enables the transparent recovery during the failover. For instance, the scheduler on the standby server will restart all jobs in its job queue with the same `jobID` assigned by the scheduler on the primary server. The jobs will be restarted on the cluster following its application specific configurations. We assume that the jobs started earlier by the primary do not interfere with the restarted jobs and will eventually terminate after the failure of the primary head node is detected. Using the mapped scheduler jobID, the status of the job in the job queue can be retrieved from the job scheduler and the appropriate output can be returned.

3.3 Event Monitoring System

The event monitoring system described previously monitors the critical system events only. Figure 2 shows the proposed event monitoring and its inter-working relationship within our framework. A scheduler wrapper will notify our event monitor for any job addition and completion in the scheduler job queue. This eliminates the need for periodic monitoring, which can lead to loss of newly added jobs if a failure occurs between two consecutive checks.

The event notification keeps the standby server always up-to-date. This also reduces the amount of processing needed in case of a large job queue. For example, in a "JOB-ADD" event, the event monitoring system notifies the job queue monitor to scan just the tail of the job queue for additions, alleviating the need to scan the entire job queue for additions and completions each time.

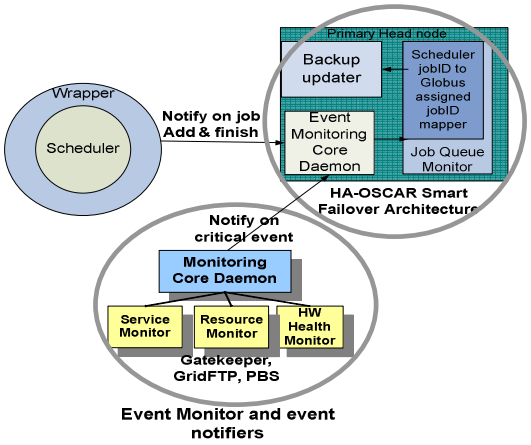

Figure 2 Proposed Event Monitoring System

4. Client & Server-Side Algorithms

4.1. Client-Side Algorithm

In our client side algorithm in Figure 3, a user submits a job using the user-specified resource (e.g. a remote site name) and resource specification language (RSL). Our failover-aware approach handles two failure scenarios: 1) a JOB_STATE_FAILED is returned to a callback function and 2) when a JOB_STATE_DONE is returned, but the stdout/stderr (output and error log) is not present at the specified location. The client has a retry mechanism based on user-specified failover duration. The relative job location derived from the GjobID is used to get the corresponding SjobID stored in it. This SjobID is used to find the status of the job in the scheduler job queue.

```
Submit job to specified resource and store the returned jobContact/jobID
If (state_of_job == JOB_STATE_FAILED ){
    sleep( for_duration_of_failover)
      dereive relative job location from jobContact
      Check status of job using the relative job location
      & the  mapped scheduler jobID
      if ( job not found in queue)
        display output // job done
      else
        display status in queue
}
else if (state_of_job == JOB_STATE_DONE){
  if (stdout/stderr not in specified location){
      sleep( for_duration_of_failover)
      dereive relative job location from jobContact
      Check status of job using the relative job location
      & the  mapped scheduler jobID
      if ( job not found in queue)
        display output // job done
      else
        display status in queue
  }
}
```

Figure 3 Failover Client Algorithm

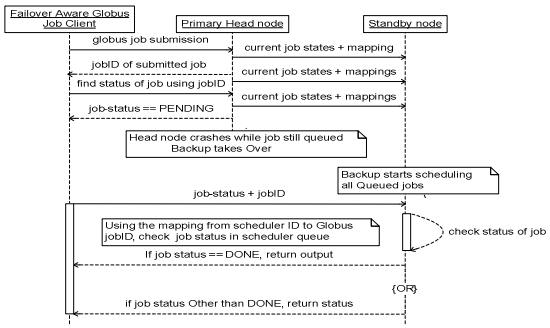

Figure 4 Event Sequence Diagram of Smart Failover Mechanism

Figure 4 illustrates a sequence diagram of the smart failover feature. Whenever an outage occurs at the head node, the standby takes over and restarts the job queue from the point of the last update.

Since a few local schedulers, such as OpenPBS, do not support checkpoint/restart recovery, our framework also addresses in-progress job fault tolerance with a reliability-aware checkpointing, an MPI-based check-pointing mechanism. Details of our checkpoint work can be found in [19]. For jobs submitted using the scheduler job submission primitives, the no modification to the submission/retrieval mechanism is needed and the scheduler would write the output/error log files to user specified directories.

4.2. Server-Side Algorithms

Figure 5 and figure 6 detail the backup updater algorithm with and without scheduler supported check-pointing. We designed our updater algorithm to support both, schedulers with checkpoint support and those without one. Hence, both algorithms have been given separately.

Our algorithms maintain two lists called *old_list* and *new_list* for updating the standby. `New_list` contains a list of jobs in queue obtained by scanning the job queue at that instant while the `old_list` is the previous list of jobs in the queue.

In the check-point-aware algorithm, we first check if every job in `old_list` is present in `new_list`, which is the latest snapshot of the job queue. If we find matching entries then we check whether they are running.

```
Update_backup_chpt_enabled (fresh_list, stored_list,
status_list){
   counter = 0
   for all Ji in old_list {
      if (Ji not in fresh_list) {
         remove  files associated with Ji from temp
         remove associated checkpoint files
       }
       else {
         increment "counter"
         if status_list[i] == 'Running'
           checkpoint the job
       }
   }
   num_new_jobs = len(fresh_list) – counter
   if num_new_jobs > 0 {
      while counter < len(fresh_list){
         if status_list[counter] != 'Running' {
           copy the corresponding job files to temp
           map scheduler jobID to corresponding globus jobID
         }
         else
           checkpoint the job
         increment counter
      }
    }
}
```

Figure 5 Backup Server Update Algorithm with Scheduler Supported Checkpointing

If the job is in `new_list` and it is running then we send a message to the scheduler to checkpoint it. If a job in `old_list` is not present in the `new_list,` then it implies that it got completed and we remove its associated files from a temporary directory, which is used to sync the backup. After comparing the `old_list` with the `new_list`, if the `new_list` has more jobs then it implies that these jobs were newly added. We proceed ahead from the job that is newer than the jobs in old list. If the job in question is queued then we copy its corresponding job files to the temporary directory that we sync up with the backup. We check whether it has been submitted through Globus job submission mechanism; if yes, then we map the `SjobID` to the `GjobID`. If the job is in a running state then we check-point it.

```
Update_backup_no_chpt(fresh_list, stored_list, status_list){
 counter = 0
 if ( new_top(stored_list[0],fresh_list) == true || new_tail
(stored_list[len(stored_list -1],fresh_list) == true )
{
 // i.e there has either been an addition or deletion
   for all Ji in old_list {
      if (Ji not in fresh_list)
         remove  files associated with Ji from temp
      else
         increment "counter"
   }
   num_new_jobs = len(fresh_list) – counter
   if num_new_jobs > 0
    {
       while counter < len(fresh_list)
       {
          if status_list[counter] != 'Running'
          {
            copy the corresponding job files to temp
            map scheduler jobID to corresponding globus jobID
          }
          increment counter
       }
    }
 }
}
```

Figure 6 Backup Server Update Algorithm with Scheduler Supported Checkpointing

In our updating algorithm where check-pointing is not supported by the scheduler, we start by comparing the head and tail jobIDs in the `new_list` with corresponding ones in `old_list`. If both of them are the same, that means that the queue has not changed and we can avoid the processing. This proves helpful when the queue length is long and processing of the whole queue will incur significant CPU processing. The remaining algorithm is similar to that with check-point support with the only difference that we do not take any action if the job is in running state. Also, as specified earlier, we do not transfer status of jobs that are in running state. So, if primary server fails while a job is running then that job is restarted on the backup as its temporary job files contain the job status as "queued" (transferred earlier). After the update algorithm finishes we replicate the *temp* directory (containing the changes in the job queue, if any), the user directories and the dataset disk (containing the datasets for jobs to run) to the backup to keep it up to date with the job related files and datasets.

5. Implementation and Experimental results

Figure 7 shows the "Task Level" fault tolerance achieved using "grid workflow" [8]. In "grid workflow", the user defines actions to be taken in a failure via a user-programmed exception handling. Our approach enhances the task-level fault tolerance by ensuring that the site remains highly available and the user can retrieve his/her output in spite of a failure.

Figure 8 shows our experimental setup. The head node was running the Redhat 9 operating system. OSCAR 3.0 was used to build a cluster and setup environment between the head node and multiple compute nodes. We overlaid Globus 3.2 on the head node; its interface to the OpenPBS job-manager was also configured. We later installed HA-OSCAR 1.0 on the head node in order to create a dual head Beowulf cluster. HA-OSCAR handles a re-establishment of NFS between the standby and the compute nodes after the failover.

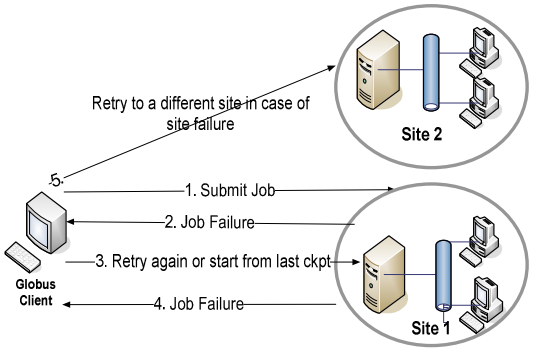

Figure 7 .Task-Level Fault Tolerance through Grid Workflow

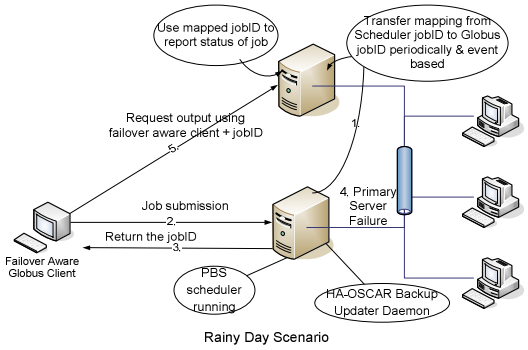

Figure 8: Experimental Setup during Failover

The job queue monitor and backup updater were running on the head node, periodically updating the standby with the critical directories and mapping from Globus GjobID to scheduler assigned SjobID (i.e. PBS in our experiment). The failover-aware client, a wrapper over

existing Globus interface (client not written from scratch), was written in Python using PyGlobus submitting MPI jobs to the PBS scheduler.

The failover aware client would take the remote machine name and the input RSL submitted to the job and failover time of the remote server. The average failover time with respect to HA-OSCAR is 20 seconds which includes time needed to clone the primary servers public and private IP, restart services such as *network*, *Xinetd*, *NFS*, *Maui* and *pbs_server* as well as resume all pending jobs on the standby head node. The 20-second delay was also accounted for *a* re-establishment of NFS between the standby and compute nodes. It is important to note that the delay introduced will depend on whether the last running jobs on primary had just started or were near completion. In the first case the delay introduced will not be substantial but in the second case, as the status of the job is "queued" at the backup, the last running jobs will be restarted, hence increasing delay for next jobs.

The event monitor as of now only triggers the job queue monitor in case of critical system event. To mimic the behavior of scheduler generated events, namely JOB_ADD, JOB_COMPLETE we generated events whenever specific temp job files got created (on job addition) and deleted (on job completion) using the File Alteration Monitor (FAM) interface. This causes the event monitor to invoke the job queue monitor to scan the head/tail of the job queue depending on the type of event generated. Whenever a job is submitted to PBS, two temporary job files are created, namely *jobid*.C.JB and *jobid*.C.SC. We used FAM to check the creation and deletion of these files to get notification of job addition and deletion.

We submitted jobs using the '*qsub*' mechanism in PBS (for cluster based job submission) and using modified failover client "grid-job-submit" for grid based submissions. In the first case (cluster based submissions), the scheduler on the standby (after failover) writes the output and error log to the specified files, enabling the remote user to connect anytime to check status of his job.

For grid based job submissions, we validated that the client was able to transparently recover from the remote failure and provided the user the correct status. The command "*grid-job-submit*" behaves similar to its Globus counterpart, only differing in a way that if the `stdout/stderr` (output and error log) haven't been specified it displays the output before exiting. When a job is submitted through '*globus-job-submit*', the output is grabbed using the '*globus-job-get-output*'.

6. Results and Analysis

In this section, we discuss and analyze results and observations during the experiments. First, we compare the total time needed for jobs submitted through scheduler primitives (not through the grid), to run with and without "smart failover". As discussed in the last section, the grid-aware HA-OSCAR failover is approx 20 seconds.

JobID	Status before failure	Without Smart Failover	With Smart Failover
1	Running	Job lost	(1.43 min + 20 sec)
2	Queued	(Based on MTTR)	(1.43 min + 2.03 min)

Table 1 Comparison of with and without Smart Failover for top two jobs in queue

Table 1 gives the comparison of with and without "Smart Failover" approaches for a job queue. Each case consists of two jobs with run times of 1.43 minutes, one running and one queued after it. In case of "Without Smart Failover" approach, if we have a failure at the head node then we lose the last running job and the queued job is resumed after Mean Time to Repair (MTTR). The MTTR could range from two min (simple reboot) to a few hours depending on the severity of the problem. In the case of the "With Smart Failover" approach, as we have the last running job in queued status on the backup, it is restarted after the failover time. The queued job is resumed after the last running job is completed.

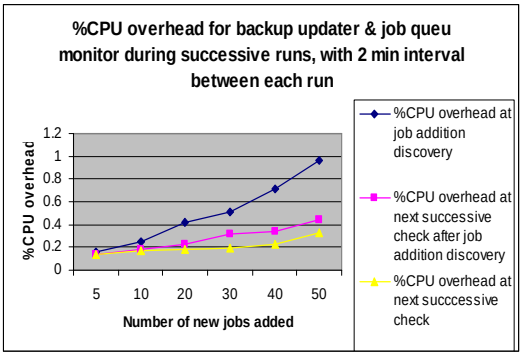

Figure 9 CPU Usage by Backup Updater and Job Queue Monitor

Figure 9 shows the percentage of CPU needed by the backup updater and job queue monitor on successive runs, with an interval of 120 seconds between each run, after new jobs were added. The number of jobs added was varied from 5 to 50 and percentage of CPU used by the program was measured using UNIX utility *time*.

In Figure 9, when job queue monitor discovers new jobs added, it incurs little CPU overhead (0.2 % - 1 %) compared to its successive runs. When 50 new jobs are added to the system, during the first run, the %CPU increases to 0.959% while during next two runs, it was 0.446% and 0.326% respectively.

For job queue replication, once a job's temporary files have been copied to the standby, the specific job update will not be done unless it completes execution. Further invocations of backup updater and job queue monitor incurred less usage of the CPU. It was observed that, only when jobs are added in burst, the backup updater and job queue monitor incurred more CPU usage.

As discussed earlier, the job queue monitor analyzes the job queue for changes and the later the backup updater replicates the changes to the standby node. The total time to update the backup with changes is composed of time taken by backup updater to replicate the changes and job queue monitor to analyze the job queue. We measured the average time taken to replicate the job queue and critical directories like the dataset disk and user directories over varying number of job additions. The number of jobs added was varied from 50 to 500 jobs. The job queue monitor and backup updater were invoked with one a minute interval between successive invocations. The total replication time needed was calculated over 10 readings when newly added jobs were discovered.

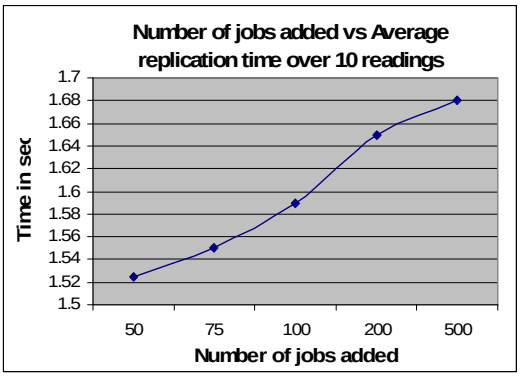

Figure 10 Comparison of Application Performance

The average time needed to replicate (via `rsync`) a burst of 50 jobs and associated directories was 1.525 seconds while it gradually increased to 1.68 seconds for 500 jobs. As can be seen from Figure 10, the average time to replicate the job queue and other critical directories increases gradually compared to the increase in the burst of jobs added.

7. Conclusion & Future Work

As cluster-based job sites increasingly become viable resources in the Grid environments, guaranteeing high availability of these job sites becomes critical in order to maximize and improve the resource utilization. There is a need to provide the site-level fault tolerance mechanism in clusters and grids to compliment the task-level fault tolerance provided by existing approaches. The earlier version of HA-OSCAR failover was enhanced with the grid-aware fault resilience in a context of the job management. As discussed, the "Smart Failover" feature in HA-OSCAR aims toward a graceful recovery by monitoring the job queue and replicating changes to it to the standby head node. The proposed event monitor would alleviate the pitfalls of periodic monitoring of job queues by triggering the job queue monitor on various critical system events as well as job addition and completion events. Hence, the standby sever is guaranteed up-to-date with a pending job queue, till the point of the failure.

The "Failover-aware" Globus job client together with the "Smart failover" feature, ensure correct job status and output in spite of a remote site failure. The mapping from the Globus assigned jobID to the scheduler assigned jobID helps the failover aware job client to retrieve the correct job status and output after failover. Our combined approaches eliminate the need for grid users to manually keep track of remote site failures and thus alleviating the need for re-submission or replication of jobs to other sites. Experimental results suggest that the "smart failover" overhead is negligible. We intend to make a production quality event monitoring subsystem in the future.

8. References

[1] Ian Foster *et al*, **"**The Anatomy of the Grid: Enabling Scalable Virtual Organizations**"**, International J. Supercomputer Applications, 15(3), 2001.

[2] Kshitij Limaye, Box Leangsuksun, *et al*, *"HA-OSCAR: Grid enabled High availability*

framework", 13th Annual Mardi Gras conference, 2005 "Frontiers of Grid Applications and Technologies".

[3] C. Leangsuksun *et al*, "A Failure Predictive and Policy-Based High Availability Strategy for Linux High Performance Computing Cluster", The 5th LCI International Conference on Linux Clusters, 2004.

[4] I. Foster and C. Kesselman, "Globus: A Toolkit-Based Grid Architecture. In The Grid: Blueprint for a Future Computing Infrastructure", pages 259–278. MORGAN-KAUFMANN, 1998.

[5] LinuxHA Clustering Project, http://www.linuxha.net//index.pl

[6] John Mugler, *et.al*. "OSCAR Clusters"**,** Proceedings of the Ottawa Linux Symposium (OLS'03), Ottawa, Canada, July 23-26, 2003.

[7] Thomas Naughton, *et al*. "The OSCAR Toolkit"

[8] Soonwook Hwang; Kesselman, C, "Grid workflow: a flexible failure handling framework for the grid", High Performance Distributed Computing, 2003. Proceedings. 12th IEEE International Symposium, 22-24 June 2003, Pages: 126 – 137.

[9] lbeaus Bayucan, Robert L. Henderson , *et al*, "Portable Batch System External Reference Specification", MRJ Technology Solutions, May 1999.

[10] Jackson, K, "pyGlobus: a Python Interface to the Globus Toolkit", Concurrency and Computation: Practice and Experience, 14 (13-15), 2002, pp. 1075-1084.

[11] J. Duell, P. Hargrove, and E. Roman, "The Design and Implementation of Berkeley Lab's Linux Checkpoint/Restart", 2002.

[12] Adding high availability to Condor Central manager, http://dsl.cs.technion.ac.il/projects/gozal/project_pages/ha/ha.html

[13] J. B. Weissman and D. Womack, "Fault tolerant scheduling in distributed networks", Technical Report CS-96-10, Department of Computer Science, University of Virginia, Sep. 25 1996.

[14] J. H. Abawajy, "Fault-Tolerant Scheduling Policy for Grid Computing systems", 18th International Parallel and Distributed Processing Symposium, 04-26-04 Santa Fe, New Mexico

[15] G. Ahrens *et al*, &#65533;Evaluating HACMP/6000: A Clustering Solution for High Availability Distributed Systems.&#65533; Proceedings of IEEE Workshop on Fault-Tolerant Parallel, and Distributed Systems, 12-14 June 1994. Pages: 2-9**.**

[16] Todd Tannenbaum, Derek Wright, Karen Miller, and Miron Livny, "Condor - A Distributed Job Scheduler", *Beowulf Cluster Computing with Linux*, The MIT Press, 2002. ISBN: 0-262-69274-0

[17] Paul Townend, Jie Xu, " Fault Tolerance within Grid environment", Proceedings of AHM2003,http://www.nesc.ac.uk/events/ahm2003/AHMCD/pdf/063.pdf, page 272, 2003

[18] Gosia Wrzesinska, Rob V. van Nieuwport, Jason Maassen, Thilo Kielmann, and Henri E. Bal, "Fault-tolerance scheduling of fine grained tasks in Grid environment", to be appeared in *International Journal of High Performance Applications*

[19] Y. Liu, C. B. Leangsuksun, "Reliability-aware Checkpoint /Restart Scheme: A Performability Trade-off", submitted to *the 2005 IEEE Cluster Computing*, Boston, MA, September 27-30, 2005

[20] R. Rabbat, T. McNeal, and T. Burke,;A High-Availability Clustering Architecture with Data Integrity Guarantees.&#65533; Proceedings of the 2001 IEEE International Conference on Cluster Computing, 2001. Pages: 178-182.