Planning-based Scheduling for SLA-awareness and Grid Integration*

Dominic Battré and **Matthias Hovestadt** and **Odej Kao**
Technical University of Berlin, Germany
{battre,maho,okao}@cs.tu-berlin.de

Axel Keller and **Kerstin Voss**
Paderborn Center for Parallel Computing
University of Paderborn, Germany
{kel,kerstinv}@upb.de

Abstract

Service level agreements (SLAs) are powerful instruments for describing all obligations and expectations in a business relationship. It is of focal importance for deploying Grid technology to commercial applications. The EC-funded project HPC4U (Highly Predictable Clusters for Internet Grids) aimed at introducing SLA-awareness in local resource management systems, while the EC-funded project AssessGrid introduced the notion of risk, which is associated with every business contract. This paper highlights the concept of planning based resource management and describes the SLA-aware scheduler developed and used in these projects.

Introduction

In the academic domain Grid computing is well known, if not even established. Researchers are using Grid middleware systems like Unicore or the Globus Toolkit to create virtual organizations, dynamically sharing the transparent access to distributed resources. Grid computing started under the solely technical question of how to provide access to distributed high performance compute resources. Thanks to numberless projects and initiatives, funded by national and international bodies worldwide, Grid systems have significantly evolved meanwhile, making Grid technology adoptable in a large variety of usage scenarios.

Companies like IBM, Hewlett Packard, and Microsoft have recognized the potential of Grid Computing already in the early days of the Grid development, providing noticeable efforts on research and the support of research communities. However, the Grid did not really enter the commercial domain until the present day. Already in 2003 the European Commission (EC) convened a group of experts to clarify the demands of future Grid systems and which properties and capabilities are missing in current existing Grid infrastructures. Their work resulted in the idea of the Next Generation Grid (NGG) (Priol & Snelling 2003; Jeffery *(edt.)* 2004; De Roure *(edt.)* 2006). This work clearly identified that guaranteed provision of reliability, transparency, and Quality of Service (QoS) is an important demand for successfully commercialize future Grid systems. In particular, commercial users will not use a Grid system for computing business critical jobs, if it is operating on the best-effort approach only.

In this context, a Service Level Agreement (SLA) is a powerful instrument for describing all expectations and obligations in the business relationship between service consumer and service provider (Sahai *et al.* 2002). Such an SLA specifies the QoS requirement profile of a job. At the Grid middleware layer many research activities already focus on integrating SLA functionality.

The EC-funded project BeInGrid (Business Experiments in Grid (BeInGrid), EU-funded Project) aims at fostering the commercial uptake of the Grid. BeInGrid encompasses numerous business experiments, where Grid technology is to be introduced to specific business domains. Successful experiments reached the goal of proving the benefit of applying Grid technology for commercial customers. According to the NGG, a major objective in these BeInGrid experiments is the provision of reliability as contractually expressed in negotiated SLAs.

Current resource management systems (RMS) are working on the best-effort approach, not giving any guarantees on job completion to the user. Since these RMS are offering their resources to Grid systems, Grid middleware has only limited means in fulfilling all terms of negotiated SLAs.

For closing this gap between the requirements of SLA-enabled Grid middleware and the capabilities of RMS, HPC4U (Highly Predictable Cluster for Internet-Grids (HPC4U)) started working on an SLA-aware RMS, utilizing the mechanisms of process-, storage- and network-subsystems for realizing application-transparent fault toler-

*This work has been partially supported by the EU within the 6th Framework Programme under contract IST-031772 "Advanced Risk Assessment and Management for Trustable Grids" (AssessGrid) and IST-511531 "Highly Predictable Cluster for Internet-Grids" (HPC4U)

ance. As central component of the HPC4U project the RMS OpenCCS has been selected, since its planning based nature seemed to be well-suited for realizing SLA-awareness. Within the project all features required for SLA-awareness and SLA-compliance have been developed, e. g. an SLA-aware scheduler, mechanisms for transparent checkpointing of parallel applications, or the negotiation of new SLAs.

The HPC4U project will end 2007. The outcome of the project allows the Grid to negotiate on SLAs with the RMS. The RMS is only allowed to accept a new SLA, if it can ensure its fulfillment. For this, the RMS provides mechanisms like process and storage checkpointing to realize fault tolerance and to assure the adherence with given SLAs even in the case of resource failures. The HPC4U system is even able to act as an active Grid component, migrating checkpointed jobs to arbitrary Grid resources, if that allows the completion of the job according to its SLA.

In this paper we first highlight the concept of planning based resource management, a fundament for realizing SLA-aware RMS. The main part of the paper focuses on the specific demands of different job types on the scheduling as well as the scheduling impact of a Grid integration. The paper ends with an overview about related work and a short conclusion.

Planning Based Resource Management

Compute clusters have a long tradition beginning in the early 1970s with the UNIX operating system (Pfister 1997). Since then many resource management systems evolved, bringing functionality targeted to their specific usage domain, e. g. capabilities on load balancing. Classic systems are mostly used in capacity computing environments, computing large amounts of data in time uncritical context.

Most of the resource management systems available today can be classified as *queuing based systems*. The scheduler of these RMS is operating one or more queues, each of them with different priorities, properties, or constraints (e. g. high priority queue, weekend queue) (Windisch *et al.* 1996). Each incoming job request is assigned to one of these queues. The scheduling component of the RMS then orders each queue according to the strategy of the currently active scheduling policy. A very basic strategy is FCFS (First Come, First Served), assigning resources to jobs according to the job's entry time into the system. Modern RMS are also using priority queues, reflecting the status of the particular jobs. However, resources are assigned to jobs from the queue head, if the system has enough free resources. If this results in idle resources, backfilling strategies can be applied for selecting matching jobs from one of the queues for immediate out-of-order execution.

Many different strategies on backfilling have evolved, each optimizing according to a specific objective or usage environment. Commonly known strategies are conservative and EASY backfilling. Both strategies only differ in their way of selecting jobs for backfilling. While conservative backfilling demands that the backfilled job may not delay other waiting requests (Mu'alem & Feitelson 2001), EASY backfilling only demands the queue head's jobs not to be delayed (Lifka 1995). For deciding about the impact of a backfilling decision on the delay of jobs in the queues, the system has to have runtime information of these jobs. Hence, specific backfilling strategies (like EASY and conservative backfilling) can only be applied to environments where these statements are available.

By switching the focus from classic high throughput computing to computation of deadline bound and business critical jobs, also the demand on the RMS and its scheduler component changes. If negotiating on service level agreements, the system has to know about future utilization, i. e. whether it is possible to agree on finishing the new job as requested.

Planning is an alternative approach on system scheduling (Hovestadt *et al.* 2003). In contrast to queuing, planning does not only regard currently free resources and assigns them to waiting jobs. Instead, planning based systems also plan for the future, assigning a start time to all waiting requests. This way a schedule is generated, encompassing all jobs in the schedule. Having such a schedule available, the system scheduler is able to determine which jobs are scheduled to be executed at what time. Table 1 depicts the most significant differences between queuing and planning based systems.

A prerequisite for planning based resource management system is the availability of run time estimates for all jobs. Without this information the scheduler has no means to decide how long a specific resource will be used by a job. Hence, the scheduler could not assign a start time to jobs following in the schedule. In case the user underestimated the runtime, the system can try to extend the runtime of this job. If this is not possible because other jobs are scheduled on the resource, having a high priority so that they cannot be pushed away, the job has to be terminated or suspended in order to have the resources available for other jobs. This may be considered as a drawback of planning based resource management. A further drawback regards the cost of scheduling. The scheduling process itself is significantly more complex than in queuing based systems.

The novel approach on scheduling in planning based resource management systems allows the development of new scheduling policies and paradigms. Beside the classic policies like FCFS, SJF (Shortest Job First), or LJF (Longest Job First), novel policies could help to realize new objectives or new functionalities. We are convinced that planning based resource management is a good starting point for realizing SLA-awareness.

Scheduling for Typical Scenarios

In this section typical scenarios will be described. Starting with the submission of a regular local job, the degree of service quality will increase with each scenario. For realizing SLA-awareness in the EC-funded projects HPC4U and AssessGrid, the resource management system OpenCCS has been used. OpenCCS is a planning based resource management system developed at the University of Paderborn. Details on OpenCCS can be found in (Keller & Reinefeld 2001).

	queuing system	planning system
planned time frame	present	present and future
reception of new request	insert in queues	replanning
start time known	no	all requests
runtime estimates	not necessary[1]	mandatory
reservations	difficult	yes, trivial
backfilling	optional	yes, implicit
examples	PBS, NQE/NQS, LL	CCS, Maui Scheduler[2]

[1] exception: backfilling

[2] Maui may be configured to operate like a planning system (Jackson, Snell, & Clement 2001)

Table 1: Differences of queuing and planning systems (Hovestadt *et al.* 2003)

Local Job Submission

The local job submission is the classic case of job submission, where a user connects locally to the resource management system and submits a new job. Since OpenCCS is planning based, it requires all users to specify the expected duration of their requests. The OpenCCS planner distinguishes between *Fix-Time* and *Var-Time* resource requests. A *Fix-Time* request reserves resources for a given time interval. It cannot be shifted on the time axis. In contrast, *Var-Time* requests can move on the time axis to an earlier or later time slot (depending on the used policy). Such a shift on the time axis might occur when other requests terminate before the specified estimated duration.

The Planning Manager (PM) is a central component of the OpenCCS architecture, responsible for computing a valid, machine-independent schedule. Likewise, the Machine Manager (MM) is responsible for machine-dependent scheduling. The separation between the hardware independent PM and the system specific MM allows to encapsulate system specific mapping heuristics in separate modules. With this approach, system specific requests (e. g. for I/O-nodes, specific partition topologies, or memory constraints) may be considered. One task of the MM is to verify if a schedule received from the PM can be realized with the available hardware. The MM checks this by mapping the user given specification with the static (e. g. topology) and dynamic (e. g. PE availability) information on the system resources. Since OpenCCS is a planning-based RMS, the PM generates a schedule for both current and future resource usage. Therewith it supports classic scheduling strategies like FCFS, SJF, and LJF, considering aspects like project limits or system wide node limits. The system administrator can change the strategy during runtime.

The PM manages two lists while computing a schedule, which are sorted according to the active policy.

- The *New list(N-list)*: Each incoming request is placed in this list and waits there until the next planning phase begins.
- The *Planning list(P-list)*: These jobs have already been accepted by the system. The PM takes jobs from this list to generate the system schedule.

The PM first checks if the N-list has to be sorted according to the active policy (e. g. SJF or LJF). It then plans all elements of N-list. Depending on the request type (*Fix-Time* or *Var-Time*) the PM calls an associated planning function. For example, if planning a *Var-Time* request, the PM tries to place the request as soon as possible. The PM starts in the present and moves to the future until it finds a suitable place in the schedule.

Figure 1 depicts a typical schedule situation in a planning-based RMS. If a user submits a new job request, the system is able to match the request properties with the current schedule, i. e. the PM and MM components of OpenCCS are checking whether it is possible to generate a new valid system schedule. In this case, the user's job request is accepted, directly returning the time when the job will be allocated at the latest. If the request cannot be realized (e. g. because the user requested for a time slot with insufficient available resources), the job is rejected. In this situation, the user can query the system for the earliest possible time to start the job request.

Deadline bound Jobs

Deadline bound jobs have to be completed until a specific time at the latest. A classic example for such a deadline bound job is a weather service which has to complete the computation of a weather forecast until 5am, since the forecast is to be broadcasted on TV at 6am. However, deadlines are also of particular importance for executing workflows, where the workflow is executed in multiple branches in parallel and where the result needs to be joined until a given time, so that also the overall workflow result can be delivered in time.

From the resource management system's point of view, a deadline bound job is a *Var-Time* resource requests. The user has to provide three key parameters:

- the number of required resources
- the duration of job execution
- the deadline for job completion

The deadline bound job is a specific case of a *Var-Time* resource request, since it may not shift arbitrarily on the time axis, but only within the boundaries given by the earliest possible start time and by the deadline. This constraint has to be regarded during the scheduling process, assigning resources early enough to allow the job to complete in time.

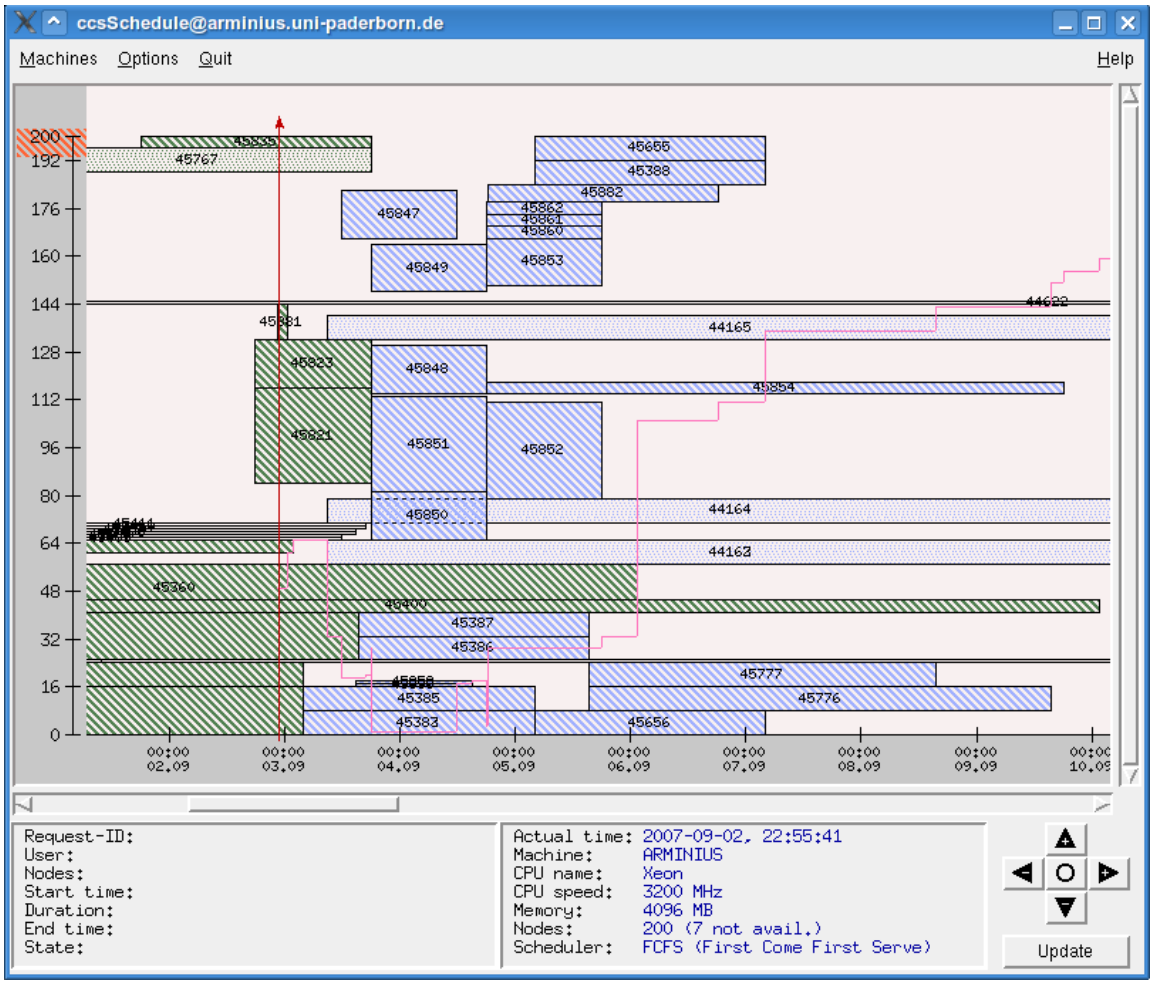

Figure 1: Schedule in a planning based RMS

At this, the latest time for resource allocation conforms to the specified deadline minus the user's specified runtime.

In the case of deadline bound jobs, the correctness of the estimated runtime of the job is crucial for the fulfillment of the deadline. It is in the responsibility of the user to give a correct estimate. If the provider assigns a resource at the latest possible start time, it is the user's responsibility if the job did not complete in time, because he underestimated the job's runtime. However, users tend to overestimate the runtime of their jobs to prevent such a situation. Hence in the typical situation the job ends long before the estimated (and scheduled) end of time. Generally assuming the specified runtime to be overestimated allows to postpone the point of latest ressource allocation by the assumed amount of overestimation. However, this strategy is risky since jobs with correctly estimated runtimes will not be able to finish until their deadline.

Due to the nature of deadline bound jobs, the scheduler has to place them after placing all *Fix-Time* resource requests, but before placing regular *Var-Time* resource requests. At this, it follows the main scheduling policy, e. g. FCFS. The scheduler executes the following steps on an initially empty schedule, trying to place *Var-Time* resource requests at the earliest possible place in the new schedule:

1. sort all requests according to the current policy
2. place all *Fix-Time* resource requests (from first P-list, then from N-list)
3. place all deadline bound *Var-Time* resource requests (first from P-list, then from N-list)
4. place all remaining *Var-Time* resource requests (first from P-list, then from N-list)

Placing deadline bound *Var-Time* jobs according to policies like FCFS does not always result in a good schedule quality. Placing jobs in front of the schedule just because they arrived at the system at an early point of time (i. e. blocking valuable resources with this job) prevents executing other jobs with perhaps even nearer deadlines. Hence, other strategies could be applied when placing these deadline bound requests.

As an alternative approach, Deadline Monotonic Scheduling (DMS) (Audsley 1993) could be applied here, where the

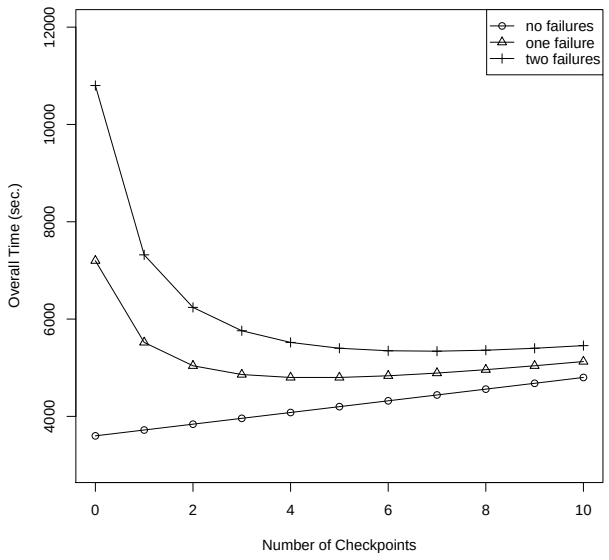

Figure 2: Impact of Checkpoint Frequency on Runtime

priority increases the nearer it gets to its deadline, i.e. the latest possible start time here. By applying Earliest Deadline First (EDF) (Buttazzo & Stankovic 1993), the scheduler would sort all deadline bound jobs by increasing remaining time until their latest possible point of start. This ensures that valuable resources are first used for urgent jobs.

Resource Failures and Fault Tolerance

A cluster system consists of multiple nodes. Partitions of these nodes are assigned to running applications, so that multiple applications are executed in parallel. If one of the nodes of a partition fails (e. g. due to a power outage), the execution of the application running on this node typically is aborted. In case of parallel applications, not only the processes of the application running on the affected node are aborted, but the entire parallel application is affected.

Cluster systems are used for speeding up the execution time of complex problems, but with an increasing grade of parallelism and an increasing runtime of the job (due to the complexity of the problem), also the possibility of a job crash increases, because only one of the nodes has to fail during the execution. This is a real problem for jobs running on dozens or hundreds of nodes over multiple days or weeks.

In the EC-funded project HPC4U mechanisms have been developed for transparently checkpointing parallel applications, i. e. all mechanisms can be applied without any modification of the job or relinking of the binary, even without having the job owner to take any notice of the mechanisms at all. This mechanism requires a patch to be applied to the Linux kernel, so that the process itself then runs inside a virtual bubble. At checkpoint time, the entire bubble is saved. For parallel applications, also the MPI implementation has to be enhanced, so that a consistent image of all parallel instances can be generated. For this purpose, the cooperative checkpoint protocol (CCP) has been developed.

Beside this stack of tools the project also evaluated other existing checkpointing solutions. At this, fairly good experiences also have been made with the tools Berkeley Checkpointing and Restart (BLCR) and LAM-MPI. Even if parallel checkpointing is possible, these tools have significant functionality drawbacks compared to the HPC4U stack.

By periodically checkpointing an application, the job can be restarted from the latest checkpointed state. Hence, only the computation steps after the latest checkpoint has to be repeated, instead of restarting the job from scratch. Even if the mechanisms have negligible impact on the job execution performance and the checkpointing of large jobs can be executed in a few seconds or minutes, this has to be considered at scheduling time.

Firstly, the effort for performing checkpoints enters the computation for the latest possible point of start. Since the time increases with the number of nodes and the amount of used memory, the system can predict quite exactly the time required for each checkpoint operation. The number of checkpoints determines the maximum time that can be lost due to a resource outage. It is a trade-off between reducing the worst-case loss of computational results and reducing the overhead of checkpointing.

The impact of the chosen checkpoint frequency on the runtime of a job is depicted in Figure 2. It assumes a job having a total runtime of one hour and a duration of each checkpoint of two minutes. The three curves represent the number of assumed resource outages. The curve depicting the case of no resource outages occurring has its minimum in $n = 0$, having no checkpoints generated. Since each checkpoint generation delays the completion of the job, each generated checkpoint is unnecessary overhead in the case of no resource outages. If no resource outages are expected or if a job restart is acceptable (like for best effort jobs), the best option is to execute without checkpoints.

In the case of resource outages occurring, things look different. An increasing number of checkpoints decreases the amount of lost compute steps lost through a resource outage, since the system is able to resume from the latest checkpointed state. The curves have their minima at the point of optimal trade-off between lost computation power and additional effort for executing the checkpoint operation. Moreover this number increases on increasing the number of expected outages. Where it is optimal to generate approximately four checkpoints in the case of one expected outage, it is approximately 7 in the case of two outages.

Secondly, the scheduling policy has to be adopted for handling the case of failures. If a job is affected by a resource outage, the entire job (not only the part of the failed node) is removed from the schedule. It leaves the P-list and is added to the *Defect list(D-list)*, encompassing all jobs affected by failures.

Then the scheduler starts the computation of a new system schedule, following the policies described above, placing jobs from D-list after jobs from P-list, but before placing jobs from N-list. This impacts new jobs (which may be re-

jected now), but does not impact other already planned jobs. However, if applying policies like DMS, the time until the job's latest point of start has to be recomputed, not taking the originally user specified job runtime into account, but the remaining runtime at the time of the last checkpoint.

The impact of resource failures on the system schedule can be reduced by introducing a failure horizon. A resource management system uses its internal monitoring mechanisms to detect problems within the cluster as soon as possible. If such a problem can not be solved by internal recovery mechanisms of the RMS itself, the cluster administrators are informed. The failure horizon represents the typical time required by administrators to solve such reported errors (e. g. 12 hours). The RMS only moves those jobs to the D-list which are planned on the defect resources within the failure horizon, assuming that the resource is available again at allocation time of all other jobs.

SLA Negotiation

The process of SLA-negotiation differs significantly from the regular job submission interface of a resource management system. There, a user submits his job description, directly getting an information about rejection or acceptance in return. In the latter case, the job has already entered the system schedule.

In case of service level agreements, a multi-phase negotiation is conducted before the job finally enters the system. The GRAAP working group (MacLaren 2003) of the Open Grid Forum (OGF) (Open Grid Forum) described such a negotiation process in the WS-Agreement Negotiation specification (Andrieux *et al.* 2004). Here the provider answers a job request with an SLA offer. The user has to commit to this offer before the SLA is actually enforced.

For the scheduling component of an RMS this negotiation process has significant implications: once the RMS has issued an SLA offer, it has to adhere to this offer until it has been committed or canceled by the user. Timeout mechanisms ensure that SLA offers automatically expire after a given time period (e. g. some seconds). However, at least during this timeout period the system has to reserve system capacity for the job in negotiation.

For this purpose, a novel list is introduced into the system: the *SLA-offer list(O-list)*. Jobs from this list are scheduled within the regular scheduling process in the order P-list before D-list before O-list before N-list. It is preferable to privilege jobs from D-list than O-list, since jobs in O-list are not yet affirmative, so that the system would not actually break an SLA-contract but only an SLA-offer. Again, the general policy of handling failures is to not affect other jobs, to keep the implication of a failure as local as possible. This also implies, that given SLA-offers should be kept if possible.

Data Staging of Grid Jobs

A second significant difference between locally submitted jobs and jobs coming from the Grid is the aspect of data staging. In case of local jobs it can be assumed that all necessary job data (e. g. the application binary and all input data) are available on a local computer system, so that fast local network connections can be used for transferring the data to the compute cluster. The time necessary for this can be neglected in general. In case of Grid jobs, this so called *stage-in* process has to be executed using slow WAN connections.

For this reason, the Grid user does not only have to specify parameters like estimated runtime, number of nodes, or deadline in the negotiation process, but also the earliest time for starting. The deadline can only be met if both the computation and the stage-in can be completed until this time. Since providers are usually connected over high bandwidth connections to the Internet, the bottleneck usually is the Internet network connection of the customer. Knowing the total amount of data that needs to be staged-in, he has to estimate the time required for transferring it over the Internet. The earliest point for starting the job is the time where the SLA has been committed (i. e. when the stage-in process could start) plus the total transfer time.

As long as the schedule has sufficient free space, the job may directly start after the estimated duration of the stage-in process. Overestimating the time for stage-in is uncritical, because this would only result in having the data available at RMS side earlier than expected. In contrast, if the user underestimated the stage-in time, the RMS is unable to start the job at the planned time. This directly threatens the fulfillment of the deadline, if the runtime is estimated correctly and there is no buffer between the planned end of the job and the deadline. The RMS has two options to handle such a situation, differing significantly in their demands on system management:

1. keeping the partition available for the job, waiting the start until stage-in is completed
2. assigning other waiting jobs to the pending job's resources, executing the pending job as soon as stage-in is completed

The first option does not require any specific RMS mechanisms, since the nodes of the pending job's partition simply remain idle. As soon as the stage-in process has been completed, the RMS starts the job. Even if this option is simple and easily manageable, it has two major disadvantages. First, the job is in danger of not finishing until the planned end, since the allocation time (i. e. the estimated runtime) is running while nodes are idle. Secondly, the overall cluster utilization is impacted, because nodes run idle instead of computing jobs.

The second option solves both of these problems, since nodes are used for computing other jobs and allocation time only starts when stage-in is completed. However, this option demands the system to support preemption of jobs. For this, we again use the checkpointing mechanisms developed in the HPC4U project. Since this solution provides transparent checkpointing for parallel applications, we are able to realize preemption for parallel jobs. For preempting a job, the job is first checkpointed and then stopped.

If other jobs are started in the partition of the pending job, these jobs have to be preempted. The scheduler is now able to rebuild the schedule after:

- subtracting the already executed runtime of the preempted jobs from their estimated runtime.

- setting the end of node allocation to the minimum of specified deadline and current time plus estimated job runtime.

This way, the job would have its entire estimated runtime available, as long as the delay in stage-in is not larger than the original buffer between end of computation and deadline. It has to be noted, that the deadline compliance of the preempted jobs is not endangered, because they already executed the time that they now get started later.

Accepting or Rejecting New Job Requests

In the previous sections it has been outlined how the demands on scheduling and system management increase with demands coming from deadline support or Grid interface. However, the general procedure of accepting or rejecting new job requests remains the same.

If a resource request is submitted to the RMS, the scheduler tries to build a new valid schedule that contains this new request. In case the scheduler succeeds, e. g. if the deadline of the new job can be realized without violating any other *Fix-Time* resource request or deadline bound *Var-Time* request, the new request is accepted by the system.

Related Work

The worldwide research in Grid computing resulted in numerous different Grid packages. Beside many commodity Grid systems, general purpose toolkits exist such as Unicore (UNICORE Forum e.V.) or Globus (Globus Alliance: Globus Toolkit). Although Globus represents the de-facto standard for Grid toolkits, all these systems have proprietary designs and interfaces. To ensure future interoperability of Grid systems as well as the opportunity to customize installations, the OGSA (Open Grid Services Architecture) working group within the OGF aims to develop the architecture for an open Grid infrastructure (GGF Open Grid Services Architecture Working Group (OGSA WG) 2003).

In (Jeffery *(edt.)* 2004), important requirements for the Next Generation Grid (NGG) were described. Among those needs, one of the major goals is to support resource-sharing in virtual organizations all over the world. Thus attracting commercial users to use the Grid, to develop Grid enabled applications, and to offer their resources in the Grid. Mandatory prerequisites are flexibility, transparency, reliability, and the application of SLAs to guarantee a negotiated QoS level.

An architecture that supports the co-allocation of multiple resource types, such as processors and network bandwidth, was presented in (Foster *et al.* 1999). The Globus Architecture for Reservation and Allocation (GARA) provides ”wrapper” functions to enhance a local RMS not capable of supporting advance reservations with this functionality. This is an important step towards an integrated QoS aware resource management. In our paper, this approach is enhanced by SLA and monitoring facilities. These enhancements are needed in order to guarantee the compliance with all accepted SLAs. This means, it has to be ensured that the system works as expected at any time, not only at the time a reservation is made. The GARA component of Globus currently does neither support the definition of SLAs or malleable reservations, nor does it support resilience mechanisms to handle resource outages or failures.

The requirements and procedures of a protocol for negotiating SLAs were described in SNAP (Czajkowski *et al.* 2002). However, the important issue of how to map, implement, and assure those SLAs during the whole lifetime of a request on the RMS layer remains to be solved. This issue is also addressed by the architecture presented in this paper.

The Grid community has identified the need for a standard for SLA description and negotiation. This led to the development of WS-Agreement/-Negotiation (Andrieux *et al.* 2004).

Conclusion and Future Work

Introducing SLA-awareness is a mandatory prerequisite for the commercial update of the Grid. Consequently SLA-awareness also has to be introduced to local resource management systems which are currently operating on a best-effort approach. The EC-funded project HCP4U aims at providing an application-transparent and software-only solution of such an SLA-aware RMS, demanding for reliability and fault tolerance. The HPC4U system already allows the Grid user to negotiate on new SLAs, which will be realized by means like process-, network,- and storage-checkpointing.

In this paper we have described the requirements of various job types and their demands on an SLA-aware scheduling. In particular we addressed the implications of a Grid integration on the scheduling policies. The described scheduling rules have been implemented within the OpenCCS resource management system, which is used in the HPC4U project. Benefiting from the mechanisms of checkpointing and restart, the scheduler has proved to be well suited for executing jobs to their negotiated SLAs. Presuming that spare resources are not allocated by other SLA bound jobs, the system is able to cope with resource outages, fulfilling the SLAs of all jobs. Thanks to the transparent checkpointing capabilities, these mechanisms also apply for the execution of commercial applications, where no source code is available and recompiling or relinking is not possible. The user even does not have to modify the way of executing the job in the Grid. Hence, HPC4U reached its goal of providing transparent fault tolerance.

However, the availability of spare resources proved to be the limiting factor at restart time. If all resources of the cluster system are allocated by SLA bound jobs, the system has no means of restarting the failure affected job, thus violating the terms of its SLA.

Improving this situation is subject of currently ongoing work. Firstly, the notion of buffer nodes is introduced to the SLA-aware scheduler. These buffer nodes may only be used for executing best-effort jobs, so that outages either affect these buffer nodes or running best-effort jobs can be displaced by SLA-bound jobs that are affected by the resource outage. Secondly, the checkpoint and restart mechanisms will be used for suspending the execution of running jobs with respect to their SLA, thus freeing allocated resources for restarting outage affected jobs. Thirdly, the scheduler will actively select jobs for migration over the Grid, so that

they can be finished on remote resources according to their SLA.

The scheduler is also the fundament for work done in the EC-funded project AssessGrid. Here, the notion of risk awareness and risk management is introduced into all layers of the Grid. This implies that the scheduler of the RMS has to consider risks of SLA violations in all scheduling decisions.

References

Andrieux, A.; Czajkowski, K.; Dan, A.; Keahey, K.; Ludwig, H.; Nakata, T.; Pruyne, J.; Rofrano, J.; Tuecke, S.; and Xu, M. 2004. Web Services Agreement Specification (WS-Agreement). `http://www.gridforum.org/Meetings/GGF11/Documents/draft-ggf-graap-agreement.pdf`.

Audsley, N. 1993. Deadline monotonic scheduling theory and application. *Control Engineering Practice* 1:71–78.

Business Experiments in Grid (BeInGrid), EU-funded Project. `http://www.beingrid.eu`.

Buttazzo, G. C., and Stankovic, J. 1993. Red: A robust earliest deadline scheduling algorithm. In *3rd intl. workshop on responsive computing systems*.

Czajkowski, K.; Foster, I.; Kesselman, C.; Sander, V.; and Tuecke, S. 2002. SNAP: A Protocol for Negotiating Service Level Agreements and Coordinating Resource Management in Distributed Systems. In D.G. Feitelson, L. Rudolph, U. S. E., ed., *Job Scheduling Strategies for Parallel Processing, 8th InternationalWorkshop, Edinburgh,.*

De Roure *(edt.)*, D. 2006. Future for European Grids: GRIDs and Service Oriented Knowledge Utilities. Technical report, Expert Group Report for the European Commission, Brussel.

Foster, I.; Kesselman, C.; Lee, C.; Lindell, B.; Nahrstedt, K.; and Roy, A. 1999. A Distributed Resource Management Architecture that Supports Advance Reservations and Co-Allocation. In *7th International Workshop on Quality of Service (IWQoS), London, UK*.

GGF Open Grid Services Architecture Working Group (OGSA WG). 2003. Open Grid Services Architecture: A Roadmap.

Globus Alliance: Globus Toolkit. `http://www.globus.org`.

Highly Predictable Cluster for Internet-Grids (HPC4U), EU-funded project IST-511531. `http://www.hpc4u.org`.

Hovestadt, M.; Kao, O.; Keller, A.; and Streit, A. 2003. Scheduling in HPC Resource Management Systems: Queuing vs. Planning. In *Job Scheduling Strategies for Parallel Processing: 9th International Workshop, JSSPP, Seattle, WA, USA*.

Jackson, D.; Snell, Q.; and Clement, M. 2001. Core Algorithms of the Maui Scheduler. In D. G. Feitelson and L. Rudolph., ed., *Proceddings of 7th Workshop on Job Scheduling Strategies for Parallel Processing*, volume 2221 of *Lecture Notes in Computer Science*, 87–103. Springer Verlag.

Jeffery *(edt.)*, K. 2004. Next Generation Grids 2: Requirements and Options for European Grids Research 2005-2010 and Beyond. `ftp://ftp.cordis.lu/pub/ist/docs/ngg2_eg_final.pdf`.

Keller, A., and Reinefeld, A. 2001. Anatomy of a resource management system for hpc clusters. *Annual Review of Scalable Computing* 3:1–31.

Lifka, D. A. 1995. The ANL/IBM SP Scheduling System. In D. G. Feitelson and L. Rudolph., ed., *Proc. of 1st Workshop on Job Scheduling Strategies for Parallel Processing*, volume 949 of *Lecture Notes in Computer Science*, 295–303. Springer Verlag.

MacLaren, J. 2003. Advanced Reservations - State of the Art. Technical report, GRAAP Working Group, Global Grid Forum, `http://www.fz-juelich.de/zam/RD/coop/ggf/graap/sched-graap-2.0.html`.

Mu'alem, A., and Feitelson, D. G. 2001. Utilization, Predictability, Workloads, and User Runtime Estimates in Scheduling the IBM SP2 with Backfilling. In *IEEE Trans. Parallel & Distributed Systems 12(6)*, 529–543.

Open Grid Forum. `http://www.ogf.org`.

Pfister, G. 1997. *In Search of Clusters*. Prentice Hall.

Priol, T., and Snelling, D. 2003. Next Generation Grids: European Grids Research 2005-2010. `ftp://ftp.cordis.lu/pub/ist/docs/ngg_eg_final.pdf`.

Sahai, A.; Graupner, S.; Machiraju, V.; and van Moorsel, A. 2002. Specifying and Monitoring Guarantees in Commercial Grids through SLA. Technical Report HPL-2002-324, Internet Systems and Storage Laboratory, HP Laboratories Palo Alto.

UNICORE Forum e.V. `http://www.unicore.org`.

Windisch, K.; Lo, V.; Feitelson, D.; and Nitzberg, B. 1996. A Comparison of Workload Traces from Two Production Parallel Machines. In *6th Symposium Frontiers Massively Parallel Computing*, 319–326.