Service Level Agreement aware Resource Management

Dissertation

von

Matthias Hovestadt

Schriftliche Arbeit zur Erlangung des Grades
eines Doktors der Naturwissenschaften

Fakultät für Elektrotechnik, Informatik und Mathematik
der Universität Paderborn

Paderborn, im Oktober 2006

Datum der mündlichen Prüfung:
7. Dezember 2006

Gutachter:
Prof. Dr. Odej Kao, Universität Paderborn
Prof. Dr. Franz Rammig, Universität Paderborn

Acknowledgements

The previous years at PC^2 have been the greatest time in my life. I am deeply grateful for all the support I received and the opportunity to pursue my way. They all contributed to make this thesis become reality. I would like to use this opportunity for thanking some special persons.

First of all I would like to thank my doctoral advisor Odej Kao for all his support and encouragement. He was always available when I was in need for discussion or advice. Under his lead, PC^2 became the perfect place to do research. Furthermore I am grateful to Franz Rammig for agreeing to review this thesis.

I also would like to thank Bernard Bauer. He did not only guide me through the jungle of proper financial reporting in EC-funded projects, for me he is also the soul of PC^2.

In PC^2 I found colleagues working with great team spirit and commitment. Felix Heine has not only been the world's best office mate, I am in particular grateful for all the fruitful discussions and the motivating artwork.

I am deeply grateful to Axel Keller, the father of CCS. Firstly for his technical expertise, but even more for his patience on listening and his commitment to the project. Many things would look different without him. Thanks also to my students Alexander Gretencord, Jan-Henrik Wiesner and Tobias Bettmann for their work.

The HPC4U project has been funded by the European Commission. Thanks to all European tax payers for giving me the opportunity to realize my ideas in this context. Discussions with research colleagues were stimulating and helpful for my work. Further I would like to express my thanks to Simon Alexandre for all the discussions we had.

Last but not least I would like to thank Dorit and my entire family. I am only standing on the shoulders of giants. In particular thanks to my parents for convincing me to continue to go to school as I decided to quit at the age of 7. It was worth it.

Abstract

Next Generation Grids aim at attracting commercial users to employ Grid environments for their business critical compute jobs. These customers demand for contractually fixed service quality levels, ensuring the availability of results in time In this context, a Service Level Agreement (SLA) is a powerful instrument for defining a comprehensive requirement profile.

Numerous research projects worldwide already focus on integrating SLA technology in Grid middleware components like broker services. However, solely focusing on Grid middleware services is not sufficient. Services at Grid middleware may accept compute jobs from customers, but they have to realize them by means of local resource management systems (RMS). Current RMS offer best-effort service only, thus they are also limiting the service quality level the Grid middleware service is able to provide.

In this thesis the architecture and operation of an SLA-aware resource management system is described, which allows Grid middleware components to negotiate on SLAs. The system uses its internal mechanisms of application-transparent fault tolerance to ensure the terms of these SLAs even in case of resource outages. The main parts of this work focus on scheduling aspects and strategies for ensuring SLA compliance, respectively design aspects on implementation.

Scheduling strategies significantly determine the level of fault tolerance that the system is able to provide. After presenting requirements of Grid middleware components on service qualities and a description of operation phases of an SLA-aware resource management system, intra-cluster scheduling strategies are described. Here, the system solely uses its own resources and mechanisms for coping with resource outages.

For further increasing the level of fault tolerance, strategies for cross-border migration are presented. Beside a migration to other cluster systems in the same administrative domain, the system uses also Grid resources as migration targets. For ensuring the successful restart, mechanisms for describing the compatibility profile of a checkpointed job are presented.

The concept of the SLA-aware resource management system has been implemented in the scope of the EC-funded project HPC4U. We will describe design aspects of this realization and show results from system deployments at use-case customers.

Contents

Contents

1 Introduction

The idea of Grid computing is similar to the former concept of metacomputing, but takes a broader approach. More different types of resources are joined:e. g. supercomputers, network connections, data archives, 3D-visualization devices, or physical sensors and actors. The vision is to make them accessible similar to the power grid, regardless of where the resources are located or who owns them. Many components are needed to make this vision real.

It is surely no overstatement to claim that Grid computing grows up little by little. Research on Grid computing started under solely technical aspects: how to realize the virtualization of resources, and how to use these distributed virtual resources. Meanwhile, Grid computing is widely accepted and no longer used by research institutes only. Companies like IBM, Hewlett Packard and Microsoft have recognized the potential of Grid Computing and are investing noticeable efforts on research and the support of research communities. Common goal is to attract commercial users for Grid Computing.

Research efforts on Grid Computing lead to numerous Grid middleware systems. The most prominent are UNICORE [97] and the Globus Toolkit [32]. However, current Grid architectures and implementations lack many essential capabilities, which would be necessary for a future commercial large scale Grid system. In this context, the European Commission convened a group of experts to clarify the demands of future Grid systems and which properties and capabilities are missing in currently existing Grid infrastructures. Their work resulted in the idea of the Next Generation Grid (NGG) [34, 35, 59].

The Next Generation Grid aims at supporting resource-sharing in virtual organizations all over the world, and thus to attract commercial users to use the Grid, to develop Grid-enabled applications, and to offer their resources in the Grid. Mandatory prerequisites are flexibility (build virtual organizations on demand), transparency, security, predictability, and reliability in communication and cooperation (Fault Tolerance), and finally the application of reliable contractual agreements to guarantee the desired and negotiated Quality of Service (QoS). Applications in these NGGs will demand the Grid middleware for mechanisms to enable a flexible negotiation of specific levels of Quality of Service. In this context, a QoS guarantee may range from the reservation of resources for a given time span, which is required for the orchestrated usage of distributed resources, up to guarantees for an advanced level of Fault Tolerance.

The guaranteed provision of reliability, transparency and QoS are important

demands of the NGG. Commercial users will not use a Grid system for computing business critical jobs if this Grid system is operating on the best-effort approach only. The user must be able to rely on getting the requested QoS level, not only meaning the predictable operation of a single resource, but also the orchestrated execution of an entire workflow.

In this context, a Service Level Agreement (SLA) is a powerful instrument for describing a job's requirement profile. It is the exact statement of all obligations and expectations within the business partnership between the resource provider and the Grid user as its customer [2]: it describes which resources should be provided in what amount for how long and in which quality. It also encompasses the price for resource consumption, respectively the penalty fee for violating the agreement.

Scientific and engineering applications in domains such as energy, CAE (Computer Aided Engineering), bio-informatics, weather modeling, pharmaceutical, automobile, fluid dynamics, and finance to name but a few form part of a widening range of computational and data intensive applications on production clusters. All these domains of application rely on a guaranteed level of Quality of Service (efficiency, predictability, scalability, and reliability) from the underlying computer architectures and from the applied Grid middleware.

These requirements lead to new claims in all Grid middleware components, local resource management systems, and in the underlying computer, storage and networking architectures. Many research projects already focus on SLA functionality within the Grid middleware. However, at present none of the processing levels (computer architectures, resource management systems, Grid middleware) complies with these high grade requirements.

It is not sufficient to add SLA mechanisms like negotiation or monitoring to Grid middleware systems only. As Grid middleware systems base on local Resource Management Systems (RMSs) for execution of Grid jobs, also these RMSs have to be able to guarantee the contents of a negotiated SLA. Comparing the capabilities of current RMS on the one side, which are at best able to reserve resources for a fixed time span, and the requirements of future Grid systems on the other, a gap between both sides becomes apparent.

1.1 Scope of this Work

The inadequacy of current systems for emerging Grid requirements has been underlined by MacLaren et al. in [55]: neither the best-effort approach of batch schedulers nor the inflexible nature of advance reservation is suitable for future resource management systems. Mechanisms are required which support SLAs, "negotiated between the client (user, superscheduler, or broker) and the scheduler". Unlike other approaches for providing SLA-awareness and service quality

guarantees, MacLaren et al propose the development of novel RMS scheduling mechanisms, which are able to negotiate with SLA-requesting customers connecting over the Grid infrastructure. Similar ideas and demands also have been presented in [27].

The first step in realizing an SLA-aware resource management system therefore is the provision of an SLA negotiation interface. This interface empowers Grid middleware components to start direct negotiations on resources, not limited to functionalities of wrapper scripts mapping requests to legacy RMS commands. This would furthermore enable broker scripts to act as mediators, matching customer requests to actually provided services at resource level.

Adding negotiation capabilities however is not sufficient for realizing actual SLA-awareness. The RMS also has to pay attention to the implications of an agreed SLA in its system management. Therefore new system management approaches are required, exceeding the currently existing best-effort mechanisms. An important task in this scope is the consideration of resource outages as a normal event in system management. By agreeing on SLAs, the system has to be able to cope with exceptional situations, aiming for adherence with all agreed SLAs.

Transparency is a central objective of Grid computing. Similar to the power grid, the user should be able to connect to his Grid computing infrastructure, consuming the required amount of compute power, not knowing or caring where the actually used resources are located or operated. The task of Grid middleware is to abstract from technical details. Transparency also is a crucial demand on mechanisms realizing this fault tolerance. The user should not have to care about details on realizing fault tolerance. He should still be able to submit his job, without relinking the application to any special purpose libraries. In fact, relinking or recompiling is not possible in many cases, because commercial users bring their commercial codes with them.

Mechanisms addressing this fault tolerance have to cover the entire process environment. Therefore it is not sufficient to solely apply process checkpoint mechanisms. The resource management system has to further regard storage aspects as well as network aspects. This way the consistency at job restart can be ensured.

The SLA-aware scheduler is in charge of assigning available system resources, so that even in case of resource outages the SLAs of affected jobs are fulfilled. To increase the level of fault tolerance, the system should actively use its environment, not only using internal resources for job restart, but also migrate to resources on other clusters or even within the Grid.

1.2 Document Structure

This thesis describes the architecture and operation of such an SLA-aware resource management system. The following chapter 2 on foundations addresses basic questions on Grid computing, resource management, and service level agreements. The main chapter 3 then explains the operation of an SLA-aware resource management system. In particular this chapter will highlight scheduling and migration aspects. In the scope of the EC-funded project HPC4U, large parts of this work have already been implemented. Chapter 4 is devoted to design aspects of this implementation. The following chapter 5 then describes results of this work, gained from practical experience on using the system by partners and other interested parties within the HPC4U context. This chapter will also address new perspectives for future developments which have emerged from this practical usage. Before concluding the work in chapter 7, chapter 6 will give an overview about related work in Grid middleware and subsystem level, which is relevant in the context of SLA-aware resource management. Approaches and results presented in this thesis have partly already been published in conference proceedings [67, 5, 27, 37, 38, 8, 39, 42], journal papers [66, 43], and a book chapter [38].

2 Foundations

2.1 Resource Management

Compute clusters have a long tradition beginning in the early 1970s with the UNIX operating system [33]. 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 high throughput 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) [23]. 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 common strategy is FCFS (first come, first served), assigning resources to jobs according the job's entry time into the system. Resources are assigned to jobs at the queue head, if the system has sufficient 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 [70], EASY backfilling only demands the queue head jobs not to be delayed [65]. 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 bounded 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. if it is possible to agree on finishing the new job as requested.

Planning is an alternative approach on system scheduling [67]. 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. The following table depicts the most significant differences between queuing and planning based systems.

	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	not possible	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 [58]

Table 2.1: Differences of queuing and planning systems [67]

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 of deciding 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 prolongate the runtime of this job. If this is not possible due to other jobs, the job has to be terminated or suspended 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, because the scheduling process itself is significantly more complex than in queuing based systems.

The novel approach on scheduling in Planning based resource management systems allow 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 optimize for new objectives or realize new functionalities. We are convinced that planning based resource management is a good starting point for realizing SLA-awareness.

2.2 Grid Computing

Despite the fact that processors get faster and faster every year, compute power has always been a precious good. Hence, the idea of collaborative usage of distributed resources is neither new nor has it been introduced by Grid computing. Already in the 1960s the Multics project [100] (Multiplexed Information and Computing Service) aimed at providing compute power to large user communities, accessing the system from somewhere remote.

In 1962 the United States Department of Defense started developing the ARPANET (Advanced Research Projects Agency Network) [85]. ARPANET was designed as a decentralized network. Thanks to the introduction of packet switching, this network was robust against outages of connected stations. In 1981 the TCP/IP protocol was used for the first time in NSFNet, a network established by the United States National Science Foundation (NSF), connecting multiple US-American universities. Nowadays, this introduction is commonly considered as the birth of the Internet and led to an incomparable integration of resources worldwide, which continues until the present day.

Not only the total number of nodes increased, but also the transfer speed of network interconnect. In the mid 1980s the NSF and later the Defense Advanced Research Projects Agency (DARPA) started funding US-national testbeds, like Aurora, Blanca, and Casa [13]. The goal of these projects was research on very high speed networks. Within these testbed projects the term Metacomputing was created. Nowadays this term is ascribed to James Catlett and Larry Smarr, founding director of the National Center for Supercomputer Applications [73], member institute of one of these testbed projects. Both expressed their ideas on Metacomputing in an article in the Communications of the ACM journal in 1992 [93]:

> While the national power, transportation, and telecommunications networks have evolved to their present state of sophisticated and ease of use, computer networks are at an early stage in their evolutionary process.

and

> The computing resources transparently available to the user via this networked environment have been called a metacomputer.

The basic idea of metacomputing is to harness the compute resources which are available within a network environment, not only limited to a single site, but potentially spanning over the entire Internet. As a matter of course, even without having metacomputers available, users can access all networked resources to compute their jobs. However, this is not what happens in practice. For each

resource the user has to know about system specific properties like location of libraries and compilers or access commands for the local resource management system. He also has to apply for local logins and know about characteristics of local system policies. For the user this implies significant additional workload, preventing the efficient usage of large infrastructures. In consequence, users only use a small subset of available resources following their subjective best-practice experiences.

2.2.1 First Generation

The vision of metacomputing was to tackle this obstacle which prevents effective usage of resources distributed over a network. Compute power should be made accessible similarly to the power grid, regardless of where the resources are located or who owns them. For satisfying his demands on compute power, a user should be able to consume the amount of compute power of other resources and finally being charged for actual consumption. Instead of dealing with system specific details, the user solely accesses a uniform middleware layer for submitting his jobs. Many components are needed to make this vision real.

Already in the 1980s researchers started to tackle Grand Challenge problems [72], i. e. key research problems requiring large amounts of compute power, typically not available at a single site. In this context, several US-funded projects started in the 1990s, using resources available in the existing testbeds. The I-WAY project (Information Wide Area Year) [22] starting 1995 focused on linking high performance computers and appropriate devices for visualization. Beside standard software installed on servers within the I-WAY network also a resource scheduler spanning over all resources has been developed. This way I-WAY provided an homogeneous layer over the heterogeneous landscape of available resources, firstly realizing ideas of metacomputing. This generation of software systems is nowadays denoted as the first generation of Grid computing.

Among the people involved in I-WAY were Ian Foster, Carl Kesselman, and Steven Tuecke, persons who influenced and impact Grid computing until the present day. To support user-level application and increase the number of applications benefiting from I-WAY infrastructure, a low-level communication layer had to be realized. Hence, the Nexus environment was adapted, such that it could be used within the I-WAY environment [47]. This adaptation represents the birth of the Globus Toolkit [29], the de-facto standard in Grid computing today.

2.2.2 Second Generation

The evolution from metacomputing to Grid computing was a smooth transition, starting in the mid 1990 and ending 1999, where a first definition and research

overview on Grid computing was given [49]. This publication also marks the beginning of the so called second generation of Grid computing. It characterizes the Grid as global infrastructure linking various types of resources. Grid middleware is placed as an additional and novel layer between the heterogeneous physical resources on the bottom side, and applications accessing these resources on the upper side, presenting them a homogeneous and uniform view on the resources.

In this time, various commodity Grid systems emerged, realizing aspects of Grid computing for their usage domain. However, also general purpose toolkits were developed.

Research on the UNICORE system (UNiform Interface to COmpute REsources) [97] has been founded by the German Ministry of Education and Research in the scope of two national projects, named UNICORE (1997-1999) and UNICORE Plus (2000-2002). The goal of UNICORE was to realize a uniform and easy to use graphical interface for submitting compute jobs to remote compute resources [53]. This system provided a certificate based security architecture, allowing the user to use various resources with a single sign-on, and securing all data transfer channels. UNICORE was minimal invasive for the local system administrator, because the system resides on-top of existing resource management systems. Beside the specification of single compute jobs, UNICORE also supports the definition of complex workflows. The development of UNICORE was also funded by the European Commission. Within the EUROGRID project, the UNICORE system has been introduced to selected target communities like biology, computer aided engineering, and meteorology [25]. Nowadays, UNICORE is known even outside Europe and used in projects all over the world.

The still dominant system in Grid computing is the Globus Toolkit [32], commonly denoted as solely Globus. As mentioned above, the first version of Globus evolved from achievements of the US-American I-WAY project. As the name indicates, the Globus toolkit does not aim at providing a monolithic system. In contrast, Globus is a collection of services that can be used by users, programmers, and applications to satisfy their demands [46]. The architecture of Globus is modular, providing services dedicated to one specific goal. This way, new services can provide higher level services by using specific services of other modules. The Globus architecture is layered, distinguishing between higher-level global services and low-level core services, e. g. communication libraries or interfaces to local resource management systems.

The first version of the Globus Toolkit (GT1) was released in 1998. It already encompassed elementary mechanisms necessary for realizing first ideas of Grid computing, e. g. the Globus Toolkit Resource Allocation Manager (GRAM), which is necessary for allocating resources provided by local resource management systems, respectively monitoring and controlling active computations. The

second version of Globus Toolkit (GT2) was released in 2002, bringing novel mechanisms like GridFTP, a file transfer protocol service facilitating existing Globus communication channels, thus also using Globus security mechanisms. A major improvement of GT2 was the Grid Packaging Toolkit (GPT)[45]. Similar to other packaging toolkits like RedHat's RPM, GPT allows Globus service developers to build packages for their software, thus significantly simplifying the compilation and installation process.

In [49] Foster gave a first definition on Grid computing, which was commonly accepted and frequently cited within the Grid community:

> A computational Grid is a hardware and software infrastructure that provides dependable, consistent, pervasive, and inexpensive access to high-end computational capabilities.

This definition is still underlining the hardware related aspects of Grid computing, even if it also lists other resource types. Grid computing follows the ideas of metacomputing, but takes a broader approach. More different types of resources are joined. The notion of a resource is explicitly not limited to hardware resources like compute nodes, supercomputers, network bandwidth, storage capacity, physical sensors and actors. It comprises also software resources (e. g. specific toolkits or libraries), information resources (e. g. data archives), or even human resources (e. g. a specialist with specific skills).

Still there is not only a single definition of Grid computing, but multiple approaches in describing its character, depending on organization background or commercialization interests. In 2002, Foster proposed a three point checklist for identifying systems as Grid systems [44]. According to this list, a Grid is a system that

> coordinates resources that are not subject to centralized control...,
> using standard, open, general-purpose protocols and interfaces...,
> to deliver nontrivial qualities of service...

Research on Grid computing started under solely technical aspects: how to realize virtualization of resources, and how these distributed virtual resources can be used. For coordinating these research efforts, people started coming together in the US-American Grid Forum in the late 1990s. In 2000, the Grid Forum merged with the European Grid Forum (eGrid) and the Asia Pacific Grid Forum, forming the Global Grid Forum (GGF) [31]. The GGF emerged as the main standardization organization for Grid computing. In beginning of 2006, the GGF merged with the business oriented Enterprise Grid Alliance (EGA) [20], forming the Open Grid Forum (OGF) [77]. However, it is not the only one. In particular, the non-profit Organization for the Advancement

of Structured Information Standards (OASIS) [76] is getting more and more impact on the evolution of the commercial Grid. This is due to their work on development, convergence, and adoption of e-business standards. But also other standardization bodies impact Grid computing, e. g. Distributed Management Task Force (DMTF) [19], Internet Engineering Task Force (IETF) [52], Object Management Group (OMG) [78], Web Services Interoperability Organization (WS-I) [79] or the World Wide Web Consortium (W3C) [101].

The first GGF meeting was held in Amsterdam in 2001 with 350 participants coming from 28 countries and more than 190 organizations. Since then, GGF met three times a year. Work within the GGF affects a large variety of topics, both technical and non-technical nature. For ensuring efficiency, more than 50 working groups evolved until the present day, each of them focusing on its own aspect of Grid computing. However, this set of working groups is not fixed. In contrast, everybody is free to use GGF meetings to raise new topics within dedicated birds of a feather (BOF) sessions. Hence, research on Grid computing is neither centrally steered, nor rigid in its structure, but highly dynamic and constantly changing.

2.2.3 Third Generation

With the second version of the Globus Toolkit as standard platform for Grid computing within the GGF, a powerful tool offering numerous services emerged. However, each deployment of this system for a specific domain required customization of services and filling of functionality gaps. Since no global roadmap existed, interfaces of services tend to be incompatible, making interoperability of independently developed services more and more complex [48, 30].

In the light of this dynamic evolution of Grid computing, the general adherence to a uniform Grid architecture serving as a blueprint for worldwide research and development was of central importance. The specification of such an architecture was the focal topic of the GGF working group on the Open Grid Services Architecture (OGSA-WG) [80]. Up to the present day, this working group is driving a continuous process of developing and improving this blueprint for a common Grid architecture. This architecture is impacting numerous working fields on Grid development, because it ranges from the description of the general taxonomy of the Grid infrastructure, up to interfaces of service elements.

The core idea of this Open Grid Services Architecture (OGSA) was the convergence between the Grid and web service technologies. The paradigm of service oriented computing had already proved itself as a powerful approach in domains like agent based computing, so this approach seemed to be predestinated for facing the challenges of the envisaged large scale Grid systems.

By specifying a common set of interface specifications, the interoperability between independently developed services on the Grid should be ensured [64].

This way it became possible to combine existing services to match given requirements as well as to exchange specific services by third party products without the risk of incompatibilities. The introduction of OGSA marks the beginning of the third generation of Grid computing.

However, OGSA is not an implementation guideline, but only a blueprint for a Grid architecture. The actual implementation is described by the Open Grid Services Infrastructure (OGSI), driven by the OGSI working group within the GGF [81]. OGSI is defining a set of WSDL specifications. These describe the interfaces, the behavior, and the schemata for services in Grid computing. A service having its interfaces and behavior defined is denoted as Grid service.

OGSI has not only been implemented in the third version of the Globus Toolkit (GT3), but also in a number of other projects. A prominent example is OGSI.NET [95], realizing the OGSI specification using the .NET platform of Microsoft.

The approach of OGSA was commonly accepted [15, 16, 86]. In contrast, in the light of emerging web service technologies, OGSI was broadly criticized. In [61] main arguments against OGSI are:

- Too much stuff in one specification: OGSI was criticized as being too comprehensive, not having a clear separation of functions.
- Does not work well with existing Web services and XML tooling: Using existing web service toolkits was problematic, because OGSI focused on XML without complying to all WSDL standards.
- Too object oriented: OGSI was realizing the modeling of stateful resources by means of web services, encapsulating the actual state of resources, not distinguishing between stateless services and stateful service entities.
- Introduction of forthcoming WSDL 2.0 capability as unsupported extensions to WSDL 1.1: OGSI followed the WSDL 2.0 standard, which was unpublished in early 2005. This complicated the usage of existing toolkits, which all still based on WSDL 1.1.

These criticism lead to the development of the Web Services Resource Framework (WSRF) [90], mainly driven by major players in Grid computing, namely Fujitsu, Hewlett-Packard, IBM, and the Globus Alliance in 2004. WSRF had to deliver the same functionality as OGSI, avoiding the listed drawbacks. Therefore, WSRF was not designed to be a monolithic architecture. Instead, WSRF consists of five specifications, which can be combined or used independently. These are:

- WS-ResourceProperties Spec describes how a WS resource can be constructed from web services and stateful resources.

- WS-ResourceLifetime Spec defines how a resource has to be deleted once its lifetime has expired
- WS-ServiceGroup Spec allows the construction and usage of groups of web services
- WS-RenewableReferences Spec enables to provide information on how to find a new endpoint reference for a service, if the current reference gets void
- WS-BaseFault Spec represents a basic error type.

In addition to these five specifications, WS-Notification allows the asynchronous notification about the state of a WS resource [61]. Since WSRF does not introduce structural modifications in WSDL or XML, it is a steps towards the convergence between Grid computing and web service technologies.

With the fourth version of the Globus Toolkit (GT4), an implementation of WSRF is available. However, there are also implementations available beside of GT4, e. g. WSRF.NET [96], an implementation of WSRF for the .NET framework of Microsoft.

2.2.4 Commercial Grids

Meanwhile, Grid computing is widely accepted and no longer used by research institutes only. Companies like IBM, Hewlett Packard and Microsoft have recognized the potential of Grid Computing and are investing noticeable efforts on research and the support of research communities.

Common goal is to attract commercial users for Grid Computing. In this context, the European Commission convened a group of experts in 2002 to clarify the demands of future Grid systems and to find out which properties and capabilities are missing in currently existing Grid infrastructures. Their work resulted in the idea of the Next Generation Grid [34]. After a revision of this document in 2004, this experts group released the third version of their report in 2006 [59].

2.3 Service Level Agreements

Grid Computing allows the virtualization of resources. Even though the notion of a resource encompasses not only hardware-resources, but also software- and information-resources, a typical application scenario is a request of resources for the execution of a computing job. This scenario will be presented within

this section to clarify the negotiation procedure. To ensure the interoperability with Grid middleware components, we apply to WS-Agreement and WS-AgreementNegotiation protocols [1] as standardized within the GGF.

Prerequisite for the negotiation process is that the service customer was able to find the resource provider. Therefore we presume, that the service provider has published some relevant information about his system, so that the service customer is interested in using the provider's resources for computation of his job. For this, it is essential that customer and provider share a common terminology how to describe resources. The publication and presentation of information about available resources as well as the matchmaking process is assured by appropriate building blocks within Grid middleware [14, 40].

We assume that the service customer spotted the resource provider and requests the provider for starting a negotiation process. The resource provider replies to this request either by denying the request immediately (e. g. because service requestor is not member of a trusted domain or because the request is not properly formulated) or by accepting the request. In this case, a negotiation template is transmitted to the requestor, which may encompass a list of service definition terms (e. g. access to locally available compute nodes) that may or may not be negotiated. For example, the provider can state that a maximum of 100 nodes may be requested or that only best-effort service is available.

The requesting customer now uses this template to formulate a request. By this, he specifies his requirements, regarding all constraints specified by the template. This request is the requirement profile of the upcoming job, specifying requirements on type of processor, amount of memory, number of nodes, network interconnect or necessary software environment (e. g. operating system, libraries, or toolkits). This request may also specify a deadline (i. e. the job should complete until tomorrow morning 8am) or the required level of fault tolerance (i. e. the system should checkpoint the job regularly).

By sending the request to the provider, the requestor starts the negotiation. The service provider checks the request for compliance with all constraints, for completeness and realizability. If the request is not complete (e. g. the requestor asks for a deadline but did not specify a runtime estimate), an answer is sent back to the customer. If the request can not be realized on the provider's system, the provider can either cancel the negotiation process, or send a counter proposal to the requestor (e. g. I can not finish the job until 8am, but until 12am). If the request is complete and can be realized on the system, the provider will accept the request.

If the provider sent a counter proposal to the customer, the requestor again checks this counter proposal for its content and either replies with a modified request, or terminates the negotiation. This procedure of request and counter proposal continues until either the request can be accepted or one of both parties aborts the negotiation process. To avoid an endless loop the number of iterations

is limited to a site specific maximum. It is also possible to limit the lapse of a negotiation process to a maximum time span.

The service customer may specify job details like URL of application, input and output files directly in his service request. In this case, the successfully negotiated job can directly be executed or planned for future execution. However, if the service request is a reservation for future execution, these details do not have to be specified at this time, because they are only required at execution time. If these specifications are not available at runtime, the RMS will not be able to execute the job, so that the reservation will expire.

The customer has the guaranty, that the provider will execute his job regarding the agreed service level specifications. When the job should then be executed at the agreed execution time, the customer will again contact the provider, requesting a new agreement. The provider will again answer this request by submitting an agreement template. The customer now refers to the already successfully negotiated agreement and adds all necessary specifications. The provider now does a lookup in his internal database for the specified agreement. If the specified request has been found and has not yet been used for the execution of a job, the new request will be accepted and executed.

This way an orchestrated execution of distributed jobs can be realized. If a job consists of multiple sub-jobs that need to be executed according to the sequence of a workflow, resources for each sub-job can be allocated by requesting appropriate agreements, even though information like URL of input and output files are not available until runtime. At the level of the resource management system, each step of a workflow is subject of a separate agreement. At the level of Grid middleware, specific workflow management blocks may orchestrate the execution of workflow jobs by initiating separate agreements with underlying layers (e. g. local resource management systems), only establishing a single agreement with the end-user. However, this is out of the scope of this document.

2.3.1 Basics

As explained in the introduction, at least the commercial user requires guaranteed service provision. Hence, if the commercial user should be attracted to use Grid environments, Grid middleware must provide appropriate mechanisms for ensuring such guarantees. However, determining if a specific service guarantee can be given to a request depends on a multitude of dynamic parameters. Therefore, service guarantees can not be published like static data (number of nodes within a cluster, amount of main memory or storage memory). Instead the requestor has to initiate a negotiation with the service provider, determining if a guarantee can be given to a request. Only if the current system situation allows assuring the compliance with the terms of a service guarantee, the provider may conclude the contract. If the a priori analysis of the terms of a requested

guarantee results in a high risk of violation, the requested guarantee should be denied.

2.3.1.1 Content of an Agreement

An agreement is negotiated between a service provider and a requesting service consumer. At this, a requesting service consumer does not have to be the end user. For improving the level of fault tolerance, the system will act as an active Grid component, requesting for spare resources within the Grid. Therefore, Grid providers can also act as resource requesters. The negotiation process is conducted using mechanisms provided by Grid middleware. If the service requestor is the end-user, he may control this negotiation process if his Grid software offers appropriate interfaces. Otherwise the complete negotiation process is transparent for the end-user.

Beside general data about both contractual parties, the agreement contains information about the service level objectives. This information represents the payload of the agreement. The contract may comprise one or more of such service level objectives, e. g. the guarantee of availability of resources in the requested amount, or at the requested level of QoS. Since the service objectives relate to the definition of a service, also the service definition itself must be part of the agreement. Concluding, the agreement encompasses at least one service definition term, referenced by at least one guarantee term. Each of these guarantee terms may be associated with a business value and violation fee representing the monetary aspects of an agreement. The guarantee term consists of at least one service level objective, defining the content of the guarantee.

An agreement may also refer to other agreements, which may be framework contracts between two parties. These framework contracts are also denoted as agreement context of the new agreement. If such a framework contract has been established, subsequent agreements must refer this framework contract for regulating specific terms within the new agreement. As an example, a resource provider may use such a framework contract to grant a special rate for resource consumption to a customer. As a matter of fact, the customer will be charged for this framework agreement.

Another important term within the agreement is the monitoring and measurement of contract fulfillment. The service consumer will most probably only agree to pay the charge for resource consumption, if the delivered service matched the terms of the agreements. In fact, customers will demand the provider to pay a penalty fee, if such a fee has been stipulated within the agreement. It is obvious that determining if a guarantee has been fulfilled can be controvertible. Hence, the fulfillment of an agreement should not only be monitored and asserted at the side of service provider and service customer, but also by a neutral third party instance.

2.3.1.2 Initiation of Negotiation

The process of service level negotiation will normally be initiated by the service customer. While this is obviously the normal case, the negotiation can also be initiated by the service provider. Provider driven negotiation may occur in situations where large amounts of resources are idle, so that the provider offers special rates for resource consumption to attract additional customers. At this, provider driven negotiation means, that a provider offers his own resources within such a negotiation process.

A service provider may also start a negotiation process for requesting the Grid for resources to improve his own fault tolerance. Even if this negotiation process is initiated by a service provider, it is not a provider driven negotiation process. In this case, the provider is in the role of a service requestor, as he requests the Grid for providing resources.

In both cases (provider initiated negotiation, respectively service customer initiated negotiation), the negotiation process on a new service level agreement does not start formless as a loose communication between service provider and service requestor. The negotiation usually starts with the transmission of a service level agreement template from the requested party to the requestor (since provider initiated negotiation is exceptional, this normally means the transmission of a template from the provider to the customer). It is noteworthy that only the structure of this template is defined (by means of a standardized schema), not the content. This template specifies the framework of an agreement and defines the aspects that can be negotiated. It may also define general rules that must be understood and followed for a successful negotiation. These general rules are also denoted as creation constraints.

2.3.1.3 WS-Agreement and WS-AgreementNegotiation Protocol

To enable both parties to perform a successful negotiation, a common protocol is required. The WS-Agreement and WS-AgreementNegotiation protocols, defined by the GRAAP (Grid resource allocation and acquisition protocol) [54, 1] working group of the Global Grid Forum (GGF) aim to match the demands mentioned above. These two protocols base on a protocol stack, essentially consisting of the protocols WS-Service Groups, WS-Resource Properties and WS-Addressing (cf. Figure 2.1).

Primary goals of these protocols are to standardize the terminology, the agreement structure and the concepts of an agreement. It also defines the types of agreement terms, the structure of an agreement template and the course of its creation, including creation constraints and required protocols for creation, negotiation and renegotiation. Last but not least, it also describes a way to express the state of an ongoing or concluded negotiation process.

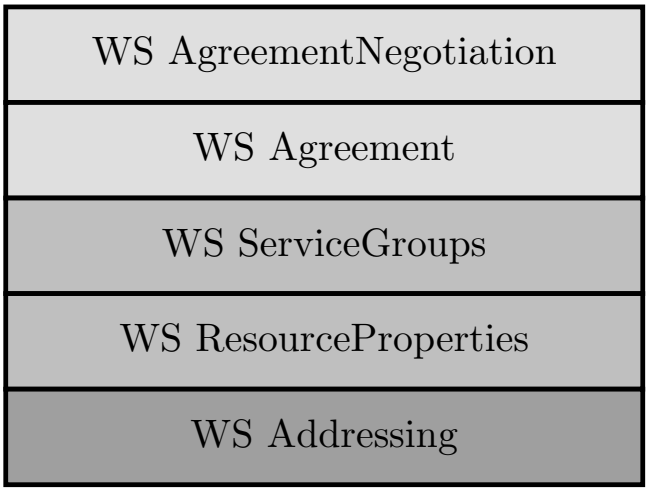

Figure 2.1: Protocol Stack

For this, the WS-Agreement protocol has to match several requirements. First, it must allow the usage of arbitrary service description terms, which implies that the protocol must not be bounded to a specific field of application. In contrast, the protocol has to be applicable in any application domain. Therefore the definition of domain specific description terms has to be supported, e. g. the definition of job specification, data service specification, or network topology specification. At negotiation phase, these predefined service description terms are referenced by the service objectives.

2.3.2 Structure of a Service Level Agreement

As mentioned in the section above, a Service Level Agreement (SLA) consists of multiple parts. Beside general data about both contractual parties, the agreement contains information about involved parties, service description terms or service level objectives. Within this chapter the structure as well as the contents of such an SLA (cf. Figure 2.2) will be explained in some more detail.

The first field of an SLA, the name, is optional. It can be used to make the identification of a single SLA easier. However, since the name of an SLA is not the unique identifier for the SLA and most SLAs are conducted and managed without human interference, the name can be left blank.

2.3.2.1 Context

More important than the name is the context of an SLA (cf. Figure 2.2). It defines general facts about the context of the agreement, like information about the parties of the agreement (e. g. name of organizations, responsible persons,

```
<wsag:Context xsd:anyAttribute>
  <wsag:AgreementInitiator>xs:anyType</wsag:AgreementInitiator> +
  <wsag:AgreementProvider>xs:anyType</wsag:AgreementProvider> +
  <wsag:AgreementInitiatorIsServiceConsumer>
    xsd:boolean
  </wsag:AgreementInitiatorIsServiceConsumer> +
  <wsag:ExpirationTime>xs:DateTime</wsag:ExpirationTime> +
  <wsag:TemplateName>xs:string </wsag:TemplateName> +
  <wsag:RelatedAgreements>
  <wsag:RelatedAgreement wsag:RelationshipType="wsag:dependency">
  <wsag:AgreementEPR>
    wsa:EndpointReferenceType
  </wsag:AgreementEPR>
  <wsag:RelatedAgreement> *
  </wsag:RelatedAgreements> +
    <xsd:any/> *
</wsag:Context>
```

Figure 2.2: XML schema of SLA context

their telephone numbers and email addresses). It is noteworthy, that an SLA may affect more than only two parties. In the case that the customer requests for a reservation of resources, the requestor may define which other parties are eligible to use the successfully negotiated agreement later on.

Optionally, the context also defines the lifetime of the agreement, which may start at negotiation time and end after job completion for normal agreements. Lifetime is of special importance for reservations and framework agreements. The context also encompasses information about other agreements that this agreement links to (e. g. in case of a framework SLA). The mechanism of SLA lifetime does not have to be used, since both parties may agree on the duration of an SLA using other mechanisms.

The context may also comprise the identifier of the agreement template, which has been used to create the agreement. The agreement identifier only has to be named if this agreement is based on special offerings related to an agreement template.

2.3.2.2 Terms

Even though SLAs are exact statements of all obligations and expectations within the business partnership between service provider and service consumer, the formulation of an SLA must be possible in arbitrary usage scenarios and application domains. Therefore the XML schema of the terms sections allows the nesting of service description terms, service references, service properties, and service guarantees. All elements can be nested and combined using logical relations AND, OR, and XOR.

The "Terms" section consists of all service description terms and guarantee terms (cf. Figure 2.3), so that this section is the real payload of an SLA. It

```
<wsag:Terms>
  <wsag:All>
    <wsag:All>
      wsag:TermCompositorType
    </wsag:All> |
    <wsag:OneOrMore>
      wsag:TermCompositorType
    </wsag:OneOrMore> |
    <wsag:ExactlyOne>
      wsag:TermCompositorType
    </wsag:ExactlyOne> |
    {
      <wsag:ServiceDescriptionTerm>
        wsag:ServiceDescriptionTermType
      </wsag:ServiceDescriptionTerm> |
      <wsag:ServiceReference>
        wsag:ServiceReferenceType
      </wsag:ServiceReference> |
      <wsag:ServiceProperties>
        wsag:ServicePropertiesType
      </wsag:ServiceProperties> |
      <wsag:GuaranteeTerm>
       wsag:GuaranteeTermType
      </wsag:GuaranteeTerm>
    }*
  </wsag:All>
</wsag:Terms>
```

Figure 2.3: XML Schema of SLA Terms Compositor

optionally encompasses service references and service properties.

Service description terms are required to identify specific services that this agreement relates to. Therefore it has fundamental importance for the agreement, as service description terms are describing the services that the agreement is about. At runtime, the provision may be bound to specific constraints and service level objectives. These objectives define how the service should be performed.

If a framework agreement is used, many requestors may reference to this agreement. Nevertheless each of these requestors may have individual definitions of service qualities and obligations. Using a service reference, that requestor can refer to the definition of an existing service instance.

```
<wsag:Variable name="CPUcount" metric="job:numberOfCPUs">
  <wsag:Location>
    wsag:AgreementOffer/wsag:Terms/wsag:All/wsag:ServiceDefinitionTerm/job:executable
  </wsag:Location>
</wsag:Variable>
```

Figure 2.4: XML Schema for defining CPU count variable

The provider may specify service properties in his agreement template. These

elements define measurable and exposed properties (e. g. amount of transferred data, runtime, or response time). Requestors may quote these service properties in the agreement request, referencing them in the service level objectives (i. e. the definition of a service level objective refers to the defined service property). In this case, the agreement has to define a variable for the referred service property. This variable can then be used in logical expressions and assertions, defining the exact meaning of a service level objective.

A sample definition of such a variable can be seen in Figure 2.4. This variable can then be used to define other variables. For example, an agreement can define the area of a job as the product of the number of CPUs used and the number of seconds used for computation. This variable "area of job" can then be used for defining service level objectives.

2.3.2.3 Guarantee Terms

A guarantee term (cf. Figure 2.5) concludes the description of all service levels that both parties agree on. These guarantee terms will be used at runtime for monitoring and evaluation purposes. Note, that an agreement may have no guarantee terms. This is the case in agreements that only offer best-effort service. However, each agreement must have at least one service description term.

```
<wsag:GuaranteeTerm>
  <wsag:ServiceScope>...</wsag:ServiceScope>*
  <wsag:QualityingCondition>...</wsag:QualifyingCondition>?
  <wsag:ServiceLevelObjective>...</wsag:ServiceLevelObjective>
  <wsag:BusinessValueList>...</wsag:BusinessValueList>
</wsag:GuaranteeTerm>
```

Figure 2.5: XML schema of a guarantee term

Such a guarantee term may guarantee the requestor that he will be provided a certain amount of nodes for a given time. The guarantee may also define the quality of service that will be provided. These statements are defined within the service level objectives.

If service descriptions are listed in the terms of the agreement, these service descriptions may be referenced in the service scope section of the guarantee terms. Then, the guarantee applies to all items of this service scope list. The usage of this service scope is mandatory, if the validity of guarantees of this agreement is beyond the validity of this agreement. This may be the case, if a framework agreement guarantees a general service level, e. g. "during the validity of this framework agreement, the system will have a maximum downtime of 60 minutes".

Agreements may be limited by qualifying conditions that have to be met. For example, a framework agreement between a customer and a resource provider can be negotiated which guarantees that the customer can always use up to 32 nodes for his computation (this is the service level objective), as long as the jobs are submitted at least 12 hours before the beginning of the requested runtime (this is the qualifying condition). Qualifying conditions can also be used for warranty exclusion.

Each guarantee term may also be complemented by business values (cf. Figure 2.6) like importance factors. They represent the importance for requestor and provider either in abstract values and/or monetary units.

```
<wsag:BusinessValueList>
  <wsag:Importance> xsd:integer </wsag:Importance>?
  <wsag:Penalty>...</wsag:Penalty>?
  <wsag:Reward>...</wsag:Reward>?
  <wsag:Preference>...</wsag:Preference>?
  <wsag:CustomBusinessValue>...</wsag:CustomBusinessValue>*
</wsag:BusinessValueList>
```

Figure 2.6: XML schema of a business value list

The definition of a penalty is of particular importance, since this property defines the amount of money that has to be paid from the service provider to the service customer, if the guarantee terms of the agreement are violated. Instead of stating a general penalty fee, which has to be paid if the guarantee terms are violated, also a dynamic penalty fee can be defined. This can be achieved by specifying an assessment interval (cf. Figure 2.7), which represents the unit for which an agreement violation will be assessed and charged. The definition of an assessment interval then has the meaning of "for every 30 minutes that the job is delayed, a penalty fee of xy EUR will be charged".

```
<wsag:Penalty>
  <wsag:AssesmentInterval>
    <wsag:TimeInterval>xsd:duration</wsag:TimeInterval> |
    <wsag:Count>xsd:positiveInteger</wsag:Count>
  </wsag:AssesmentInterval>
  <wsag:ValueUnit>xsd:string</wsag:ValueUnit>?
  <wsag:ValueExpr>xsd:any</wsag:ValueExpr>
</wsag:Penalty>
```

Figure 2.7: XML schema of assessment interval

Similar to the definition of penalty fees for violation, it is also possible to define rewards for fulfilling the guarantee terms. In that case, the service customer has to pay a basic fee for the agreement. If a certain guarantee term can be met, he is additionally charged for the reward (e. g. the service provider assures that

the customer can access up to 32 nodes for computation if the job has been submitted at least 12 hours before beginning of requested runtime; if the service provider makes it to provide resources in less than 12 hours time before the job, the customer pays an additional reward for this.).

The definition of preferences is another way of fine-grained description of business values. Preferences are of high importance during the negotiation phase, as the service requestor may define multiple alternative service requests at the same time. This enables the service provider to evaluate which service request could be realized, and to return a single answer to the requestor, which may dramatically reduce the number of negotiation iterations.

2.3.3 Negotiation of Service Level Agreements

In the previous sections the basic workflow of a negotiation as well as the contents of a negotiated Service Level Agreement (SLA) have been presented. Within this section the negotiation process itself should be focused.

In the simplest case a negotiation process consists of only a single iteration: The service requestor submits an agreement request to the service provider, who either accepts or refuses the job, depending on the requirements within the request and the current local system conditions. Since no real negotiation process takes place, it can be spoken of pseudo-negotiation here.

Non-trivial negotiation takes multiple negotiation iterations. If a service request can not be fulfilled by the service provider, a counter proposal is sent back to the service requester. Based on this counter proposal a new request is generated, starting the next iteration of negotiation. This negotiation process ends if the request is accepted by the service provider or the negotiation process is terminated due to the contents of the counter proposal respectively request, or due to a reached maximum number of negotiation iterations.

Complex negotiation does not only take multiple iterations, the service requestor also defines preferences. This allows the service provider a higher degree of freedom in resource planning, thus resulting in a higher chance of a successful negotiation.

The GRAAP (Grid Resource Allocation Agreement Protocol) working group of the Global Grid Forum (GGF) is currently working on defining a negotiation protocol named WS-AgreementNegotiation, which resides on top of WS-Agreement. It provides a simple state-machine (cf. Figure 2.8), which represents the course of a negotiation process.

A negotiation process can be initiated both by service consumer and service provider. Therefore the negotiation protocol has to be symmetric. The party that stated the negotiation process is called initiator within the state machine. Accordingly, the other party is called responder.

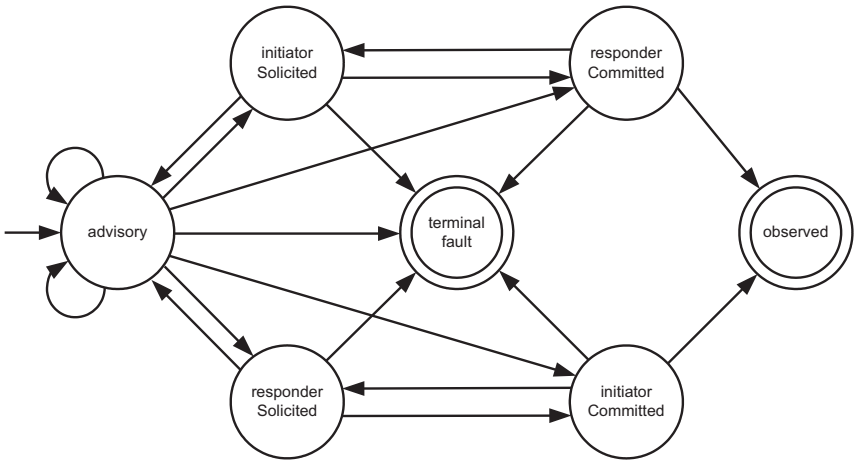

Figure 2.8: GRAAP Negotiation States

WS-AgreementNegotiation classifies the types of messages between initiator and responder as follows:

1. Advisory offers are messages between both parties without any consequences like obligations or restrictions for further actions.
2. Soliciting offers do not have an obligation as consequence, but they require the partner for a counter-offer, which can again be of arbitrary kind.
3. Committing offers imply that the sender commits himself to the terms that have been offered. The communication partner can then decide to reject or accept this offer.
4. Accepting offers indicate that the communication partner accepts the offered terms that the other partner has committed to.
5. Termination messages interrupt the negotiation process, regardless of the state of each communication partner or the contents of actual offers.
6. Reject messages only reject the last offer of the communication partner, not terminating the negotiation process.

The state machine always starts in the advisory state. The solicited state is reached, if one communication partner submits a service request. Since the solicited state requires a counter-offer, the receiving partner has to analyze the received request. Based on this process, some kind of answer is returned to the requesting partner. In case of acceptance, the requested partner changes to the committing state, waiting for acceptance from the requestor. If the partner accepts the terms, both partners change to the observed mode.

3 SLA-aware Scheduling

Fault tolerant resource management needs various components to achieve the goal of fault tolerance. Beside operative components like process checkpointing or storage snapshoting (i. e. the checkpointing of storage partitions), it is the resource scheduler which has to decide and orchestrate the usage of fault tolerance mechanisms.

However, there is not only the compute part of a job that has to be considered by a scheduler. Before a job comes to actual execution, both contractual parties negotiate on the level of service that has to be provided. Moreover, all input data is transferred from the user to the compute resource, respectively the transfer back of all result data to the user after successful execution. These operations are denoted as phases of operation and will be explained within this chapter.

Fault tolerance mechanisms enable the development of novel scheduling approaches, targeting on the demands of future commercial Grid users. The scheduler has to use these mechanisms to decide on accepting new SLAs as well as to adhere to all terms of already accepted SLAs. The focus of this chapter will be on presenting the design of such a scheduler and its application in practice. In this chapter, the additional options on scheduling and system management resulting from a Grid integration will be described.

3.1 Negotiation Requirements of Broker Services

At the Grid Middleware level, broker services are in charge of mapping non-trivial workflows (i. e. workflows consisting of more than only one workflow task) to appropriate resources. These workflows can either be predefined by the connecting Grid customer or loaded from a workflow description database. Such a database comprises knowledge about workflow details, e. g. which workflow steps are necessary to obtain a high level goal. By using such a database, a user does not have to have knowledge about how to solve a high-level goal, he only selects a matching workflow from the database, which then expands to a orchestrated workflow description of single workflow tasks. In both cases the broker service has to parse the workflow description, analyzing the requirements and dependencies of the single workflow tasks.

For each of these tasks the broker first has to query resource information services, specifying static requirements of the workflow task like number and type of resources. The information service answers with a list of service providers

potentially capable of fulfilling these resource requirements. Since this answer solely bases on static information, this does not imply a guarantee that the provider is actually able to provide the resources at the requested time in the requested quality. Hence, the broker service has to negotiate with resource providers task by task. Only if the broker service succeeded in agreeing on SLAs with resource providers for all tasks, it succeeded in instantiating the workflow on Grid resources.

A workflow is a graph, not only comprising of workflow tasks as nodes of the graph, but also of task transitions as edges between the tasks. These transitions describe the order of execution in which the tasks have to be executed. The execution may be split on a node, such that the output of a workflow task is used as input data for multiple following workflow tasks. Likewise join nodes use the output of multiple predecessor tasks as input data. The execution of a workflow therefore is not only a sequential execution of workflow tasks, but a combination of sequential and concurrent tasks.

If the workflow broker service at Grid middleware has to provide service guarantees for the entire workflows (e. g. to comply with a deadline for the overall workflow execution), it first has to negotiate with resource management systems for all workflow steps. Only if the broker can ensure the compliance with all QoS requirements (i. e. if the broker is able to find resources providing the required QoS aspects) it is able to agree on the SLA for the overall workflow.

Due to this multi-phase nature of workflow mapping at workflow broker level, the resource provider has to provide more than solely fixed SLA negotiation.

- non-binding request: if a resource broker is in charge of orchestrating concurrent tasks between resource providers, it may first need a list of potential time slots for execution of each resource provider. The broker therefore requests for a list of potential starting points. The resource provider in turn answers with such a list, however only matching the current system situation with the requirements of the new request. This does not imply the actual reservation of resources for this new request.

- preparatory request: after the resource broker identified resource providers for all tasks of the workflow, it starts on negotiating for preparatory agreements with all resource providers. Here, the requestor already negotiates on the terms of the final agreement, so that the resource provider may only agree on this agreement request if it is able to comply with all its terms. If the broker service succeeded in these preparatory negotiations with all workflow tasks, it may accept the SLA-bound workflow request, because it can ensure the availability of appropriate resources at runtime. In contrast to binding agreements, the lifetime of a preparatory agreement is limited to a few minutes. If the preparatory agreement has not been

confirmed until timeout, the resource broker voids the reservation, freeing the reserved capacity in the system schedule for other requests.

- binding request: after the broker service has confirmed a preparatory agreement, the resource provider removes the timeout from this agreement, so that the resource management system provides all system resources as specified within the agreement when the runtime of the new jobs starts.

In commercial environments the resource provider will charge the requesting user for his services. For this purpose service provider and service consumer can agree on the price for resource consumption. This price may depend on the number of provided resources and the time used on the system. As a matter of fact, different charging policies have to be applied for non-binding, preparatory, and binding requests. Where regular pricing applies to the binding request, the preparatory request should have a significantly lower price representing the resource consumption caused by the resource reservation until timeout. The requesting service is charged by this amount in case that he does not confirm the preparatory agreement in time. Without any charging the provider is endangered of fraudulently mass requests, striving to block resources, thus decreasing the utilization of the machine. Since non-binding requests do not block any resources, the requesting user does not necessarily have to be charged.

Non-binding and preparatory agreements can not only be negotiated by workflow broker services, but any requesting parties within the Grid. Even end-users are able to request, even if this functionality is primary beneficial in the workflow context, not on placing single jobs on Grid resources. Since single node jobs can be placed in one iteration, a direct binding negotiation is sufficient here.

3.2 Levels of Service Quality

An SLA-aware resource management system has to provide a fault tolerant operation which is able to agree upon the level of fault tolerance delivered to the user by the means of an SLA. This SLA negotiation is based on the protocols WS-Agreement and WS-AgreementNegotiation. Since these protocols solely specify the form of negotiation but not the content, the application of these protocols is not bound to a specific domain of applications. The usage of these protocols for a concrete application demands the definition of specific Service Description Terms (SDT). With these SDTs, the user can specify, e. g. a number of nodes, available disk space, an interconnect type.

For standardizing the execution requirements of a computational job, the Job Submission Description Language (JSDL) has been introduced by the JSDL-working group of the Global Grid Forum. By means of JSDL all parameters for job submission can be specified, e. g. name of executable, required application

parameters, or file transfer filters for stage-in and stage-out. Resource specific SDTs categorize available resources by their type and functionality.

Resource specific SDTs subjecting static information (e. g. number and type of required processors) can be resolved using standard Grid information catalogue systems. Dynamic attributes need to be directly negotiated.

Beside these resource related SDTs also QoS-related SDTs are essential for enabling the service customer to request for a given level of QoS. QoS specific SDTs describe which QoS guarantees and mechanisms have to be provided by the system for the job.

Standard JSDL does not yet cover these kind of job description parameters. However, these QoS-related SDTs can be introduced as user-specific extension to JSDL. An evaluation of these QoS enhancements to JSDL only has to be conducted by the requesting customer (e. g. the end-user connecting the Grid with his interface or the Grid broker) and the SLA-aware resource management system. This way this QoS negotiation easily integrates into standard Grid service deployments.

These QoS-related SDTs have to be further divided into SDTs relating to resource specific and job specific parameters. The first specify requirements on QoS for a given type of resource, the latter specify overall job related QoS requirements, e. g. requirements in system redundancy.

3.2.1 SDTs on Resource Specific QoS

Resource specific QoS properties describe the user's demands on specific resource properties. If jobs have particular demands on resource performance characteristics, these properties are vital for reproducibility of job performance on the Grid resource. If a job starts in such a well defined system environment, the envisaged job results can be achieved in time as planned. If the Grid resource differs significantly from the demanded QoS profile (e. g. slow network), the job may not finish within the specified time, thus resulting in a violation of the job deadline.

Examples for such resource specifc QoS SDTs are:

- compute node: checkpoint frequency
- storage: availability and type of redundancy, guaranteed performance on reading and writing of data
- network: availability of failover mechanisms, exclusive usage of network interconnect, guaranteed bandwidth, guaranteed latency

At runtime of the application, the resource management system has to ensure that these agreed parameters are granted. In case of parameters like the guaranteed checkpointing frequency or exclusivity of network access, the system has

to ensure the timely execution of actions or the proper initialization of the job execution environment.

Performance oriented parameters like performance on reading and writing of storage data, the resource management system has to use its monitoring mechanisms as well as the monitoring capabilities of subsystem components. If a violation of these parameters is detected, the system has to use its mechanisms on migration for resuming the job on new resources that are able to provide the requested level of service quality.

3.2.2 SDTs on Guarantee Level

With agreeing on job related QoS SDTs, both service requestor and service provider can agree on the appliance of specific mechanisms. Thus, the provider primarily only promises to use his mechanisms on fault tolerance in a specific way. The provider does not necessarily guarantee on finishing the job within the given deadline.

Similar to regular insurance policies known from everyday life, service provider and service consumer have to agree on the contents and implications of an SLA guarantee. While low priced policies only cover a small number of services, higher priced policies offer a more comprehensive catalogue of services. The same holds valid for the negotiation on QoS parameters. If both parties negotiate on QoS aspects like the adherence with a given deadline, it has to be clarified which type of unforeseen events the provider has to compensate, respectively which fault tolerance mechanisms he has to enforce.

In this context we distinguish between the guarantee levels Background Service, Legacy Service, Enhanced Service, and Full Service, each defining how the resource management system has to handle the job and how to apply internal fault tolerance mechanisms:

- Background Service: Providers are primarily interested in a high utilization of their resources. However, even in highly utilized cluster systems, resources run idle from time to time. This undesirable situation is due to the fact that even small jobs do not fit into the gaps of a regular schedule. If the customer is not interested in getting the results at a given time, he can choose the level of background service. Here, the provider will assign gaps within the schedule to this job.

  This service may also be combined with other services (except the legacy service). For requesting these services, the user has to specify the maximum runtime of his job. This is required for enabling the resource management system to give guarantees for fulfilling the deadline. If the job does not finish within this maximum duration, it is cancelled and removed

from the system. By combining with background service, the job is automatically converted into a background service. Hence, the user does not get his results within the negotiated deadline, but temporary results are not lost, and only the remaining compute time has to be finished. However, the probability of matching free gaps decreases with the number of requested resources, so that there is little point in requesting background service for massively parallel applications.

- Legacy Service: If legacy service is agreed, the deadline will be realized by means of advance reservations without any additional fault tolerance mechanisms. If all resources (i. e. nodes, storage, network, ...) operate without exceptions and outages, the deadline will be met. However, this service will not provide any mechanisms for handling exceptional situations.

- Standard Service: Providing this service, the system will ensure the adherence with given deadlines by means of advance reservations. For fault tolerance provision, the system will perform regular checkpoints of running jobs. In case of resource outages, the system will first check the local cluster system for spare resources. If these resources were found, the checkpointed job will be restarted on these resources. Otherwise, the system will query the Grid middleware for appropriate resources for trying to resume the job on these resources. If Grid resources are found, the job will be restarted on these Grid resources. This service does not contain a guarantee that these resources are available.

- Enhanced Service: Just like operating with standard service, the system will checkpoint running jobs regularly. In contrast to the standard service, it will always ensure that spare resources are available at the local cluster, so that the job does not have to be migrated on Grid resources. Since the provider has to be secured for cases of massive resource outages, this liability has to be limited to a maximum of concurrent resource outages.

- Full Service: In contrast to enhanced service, the system will reserve remote resources at full service, so that their availability is ensured in case of failures. Only if these remote resources can be reserved, the system will accept the SLA request.

Comprising, only with full service the customer gets unlimited guarantee. Here, the provider has to pay the contractually agreed penalty fee if the terms agreed within the SLA are violated, e. g. if he missed the deadline for job completion. In contrast, the legacy service does not include mechanisms on fault tolerance at all. It is best effort service only.

Standard and enhanced service result in limited liability of the provider. The provider only has to pay the contractually agreed penalty fee if he failed to enforce the agreed fault tolerance mechanisms (e. g. if he did not checkpoint the job regularly or if he failed to provide backup resources for restarting the job in case of resource outages). Hence, the provider has to be able to prove the fulfillment of the agreement terms, e. g. by means of system protocols.

3.2.3 SDTs on System Policies

Policies are defining the general behavior of a system. By specifying commonly accepted or locally published policies within an SLA, the mode of operation is defined at runtime. Similar to SDTs on resource specific QoS, the resource management system has to prepare the execution environment of a job prior the execution of a job as well as using its monitoring capabilities during runtime. In case, the system then has to react as specified within the policy.

- Security Policy: Security is vital for acceptance of Grid systems by users. By specifying a commonly accepted security policy, the requestor can be sure that access to his data and job is restricted in a defined way. For top secret computations such a security policy can demand the restriction of physical access to compute nodes to selected administrative staff. If the provider is not able to provide this level of security, the job may not be executed on this Grid site.

- Migration Policy: The resource management system realizes fault tolerance by means of process checkpointing. If a node fails, the job can be restarted using the last checkpoint. Usually the job can not be restarted on the same set of nodes, so that the checkpoint has to be transferred to a new set of nodes. These nodes can be part of the same cluster, but may also reside to any other cluster system. By specifying a migration policy, this process of selecting new compute resources for restart of the checkpointed job can be restricted and controlled. For high security jobs the migration policy may demand that the job is only migrated to other sites, also able to fulfill the assured security policy. The migration policy may also demand not the transfer the job to other cluster systems or not to query the Grid for spare resources. The demand of a restrictive migration policy may significantly limit the system's opportunity to cope with resource outages, interfering with the provision of full service. Here, internal evaluation mechanisms have to ensure the applicability of full service in the context of the demanded migration policy. If full service can not be granted, the request has to be rejected.

3.3 Phases of Operation

For the user or Grid customer, the computation of the job on the allocated compute resources is of greatest interest. However, other phases of operation can be identified (cf. Figure 3.1), where the system handles the job [42]. As the customer has specific requirements how his job should be executed, he initially requests a service level agreement from the system. This is the first stage of operation. At the end of this stage, both parties (i.e. the cluster system and the Grid customer) agree on the contents of an SLA, or the negotiation is aborted without agreeing on an SLA.

In the next phase, the pre-runtime phase, the validity period of the SLA has not actually started, the system has to prepare itself for this new job. This means that the network has to be configured (e. g. modifying the network routing or establishing network failover mechanisms), the assigned compute nodes have to be initialized and the storage has to be provided.

The main phase of operation is the runtime phase, which starts at the beginning of the validity period of the SLA. As a first step, necessary input data is transferred from the Grid customer to the compute resource. This process is denoted as stage-in. If all input data is available, the job can start its computation, generating temporary and result data, and using all agreed resources. During this computation, the cluster middleware has to ensure the compliance with all QoS statements of the SLA. This may imply the regular generation of checkpoints or the utilization of available fault tolerance mechanisms. If the computation has finished, result data will be transferred back to the user (i.e. stage out).

The final phase of operation is the post-runtime phase, where the validity period of the SLA has ended, the job has terminated and all output data has left the system. In this phase the system can be reconfigured to "normal" operation.

3.3.1 Negotiation

If the Grid user wants to negotiate on resource usage with the resource management system, the user first submits a negotiation request. The RMS now decides if it wants to accept this request, thus starting a Service Level Agreement (SLA) negotiation with the requesting Grid user. If it declines the request, no SLA negotiation will start. The reason for this cancellation may be manifold, e.g. if the requestor is not member of a trusted domain or if accounting is not guaranteed. This decision process will be policy driven.

In case the system accepts the request, the whole negotiation process is steered by a negotiation module within the RMS, as only the RMS has a complete sight about all resources.

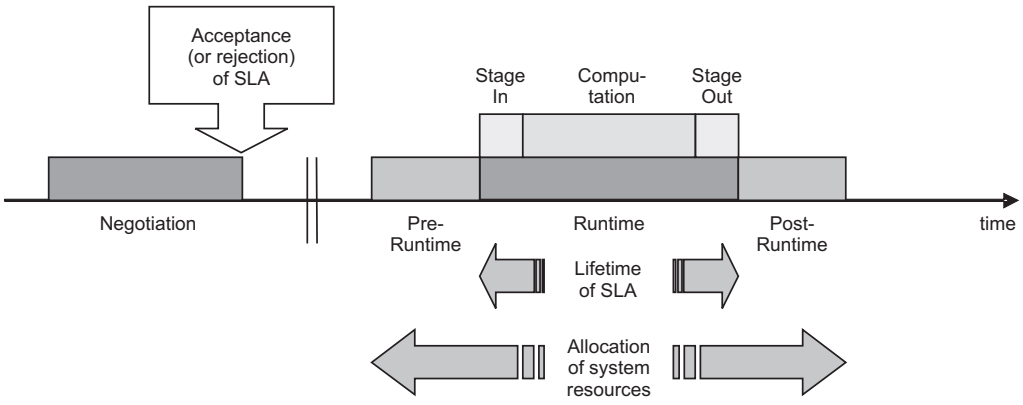

Figure 3.1: Phases of Operation

The negotiation module now initiates a new SLA negotiation with the requesting Grid user by creating a Grid service for this specific negotiation instance. This Grid service creates a specific template for an SLA and returns an answer to the requestor. This answer contains the generated SLA template as well as the endpoint reference of the recently instantiated negotiation Grid service. The SLA template represents a general framework for all further negotiation activities. It gives a formal structure for negotiation as well as Service Description Terms (SDT), which may be topic of the negotiation process.

The requestor now starts the SLA negotiation by utilizing the received SLA template. He creates an SLA request based on the received SLA template which specifies all his resource and QoS requirements. If the SLA request is complete, he transmits the SLA request document back to the system.

Now, the negotiation module of the RMS is in charge of verifying the statements of the received SLA request. This is done in a first step by checking formal requirements, e.g. constraints on available resources. If static boundaries meet existing resource limits, the negotiation module checks in a second step dynamical aspects of the received SLA request. This affects the question, whether or not the specified and requested resource is available at the requested quality at the requested time frame.

In case the requestor specified QoS parameters concerning the checkpointing of the job, e.g. a deadline bounded job, the RMS will not contact the checkpointing subsystem. The availability of checkpointing mechanisms on specific nodes of the cluster system does not depend on dynamic values. In contrast, the availability is a static property of a cluster node. Hence, the RMS can determine without any communication to the checkpointing subsystem, if checkpointing mechanisms can be provided.

It is important to stress that the specific requests on QoS can not be regarded isolated from each other. In fact, they interfere each other. In case of a deadline bounded job, the storage subsystem has to provide additional storage capacity for saving process checkpoints. QoS requests even interfere within a single subsystem: in case more than one storage snapshots shall be saved (i. e. the checkpoint of all contents of the storage partition which is used for job execution), the storage capacity has to be aligned.

3.3.2 Pre-Runtime Phase

Each negotiated SLA has a unique identifier which is used as reference in further communication. In case of a successful negotiation procedure, the RMS saves the SLA in its internal database.

Due to the SLA-aware scheduler, the contents of all negotiated SLAs are part of the scheduling process. Hence, the system awaits the incoming job and assigns appropriate resources. To utilize these resources, further communication concerning the SLA-bounded job always refers to the unique identifier of the negotiated SLA.

At the moment where the new job enters the system (this does not only imply the executable of the application that should be executed, but also all input data required by this application), the components already have to be fully prepared for this new job. This process of initialization is denoted as pre-runtime phase.

It is of great importance to consider this phase, as open tasks may be time consuming due to their complexity or communication intensity. As an example, the nodes may have to be configured to have a configuration as agreed within the SLA. This task may range from the provision of specific libraries and compilers, up to the installation of the whole node environment. It is obvious that this process needs to be considered in the scheduling process, so that the compute environment is up and running at the agreed time.

In this phase the RMS initializes the working environment of a job, concerning the configuration of local cluster nodes, the network subsystem, and the storage subsystem. The emphasis of this process will be on establishing a fault tolerant environment, according to the specifications of the SLA. All operations within this stage are started, steered and monitored by the resource management system.

3.3.3 Runtime Phase

After this first stage, the basic environment of the job has been established. Now, the stage-in of process data can proceed. Stage-in is performed by mechanisms at the level of Grid middleware, e. g. the Globus toolkit. Hence, the RMS does not have to provide these mechanisms on its own. These stage-in

mechanisms ensure that the process will find its data at runtime. The RMS only has to ensure that storage capacity is available at the agreed quality and quantity.

Once all necessary data has been staged in, the computation of the job may start. The main task of the RMS is to ensure that all resources are available as agreed in the SLA during runtime. Therefore monitoring mechanisms have to be used, checking if all resources are operating in normal mode. If resource outages occur (e. g. dropout of a compute node), appropriate failure tolerance mechanisms have to be enforced. Due to adherence reasons with all SLAs, it is of vital importance that resource failures are detected as soon as possible.

For being able to cope with resource outages, the resource management system has to execute adequate precautionary actions, e. g. checkpointing mechanisms (refer to Section 4.3 for details on checkpoint generation). In case of resource outages the RMS will then use its fault tolerance mechanisms, e. g. using generated checkpoint datasets for restarting the job. Presuming the availability of monitoring facilities, the system can use its fault tolerance mechanisms also in a precautionary manner, e. g. reacting on anticipated failures instead of waiting for actual resource outage.

The primary goal of all fault tolerance mechanisms is the successful completion of a job. The result of a successfully completed job normally is a result dataset. This dataset has to be transferred back to the Grid customer, the owner of the completed job. This transfer is done in the stage-out process, which concludes this stage. Just like the stage-in process, the stage-out process will be performed by mechanisms of Grid middleware.

3.3.4 Post-Runtime Phase

The post-runtime phase is the last step of resource consumption. At this point, the computation of the job has finished and all result data has been transferred back to the service client. This stage is the counterpart of the pre-runtime stage, since specific configuration of the cluster system may have to be revoked. This reconfiguration does not only affect the configuration of the compute nodes, but also the configuration of the storage or network subsystem. Furthermore checkpoint/snapshot datasets can be removed, since the job has been completed and these datasets are not required anymore.

Another important task of the post-runtime stage is the analysis of the job runtime. As all monitoring data is available at this point, a concluding analysis of these logs can be accomplished. Goal of these checks is to determine if all specifications of the SLA have been fulfilled. In case of resource outages, it has to be checked if the resource management system has reacted as agreed.

3.4 Timing Aspects of Runtime Phases

In every runtime phase the resource management has to fulfill a given set of operations [43]. These are vital for providing the agreed service level to the running operation. Since these operations are consuming a specific amount of time, this needs to be considered at negotiation, scheduling and general system management.

3.4.1 Overhead caused by Initialization

In the pre-runtime phase, the assigned cluster partition has to be prepared for the job which is about to start. This initialization can range from configuration of system services up to installation of additional drivers and modules or even the boot of a different operating system.

Resource management systems offering fault tolerant service may enter this state in two cases: firstly, the job is about to start for the first time. This is the classical case, which is also occurring in standard RMS. Secondly, the job may be affected by a resource outage and is now subject to restart. Here the RMS has to prepare the partition of the cluster such that the job can restart successfully from the latest checkpointed state. Both cases differ significantly in their tasks and therefore also in the time consumption.

The catalogue of tasks for initialization of newly starting jobs depends on the amount of properties that the RMS negotiating on. If the requesting user is able to request for a specific job environment (e. g. availability of libraries in a specific version), the RMS has to perform all necessary installation tasks so that the node environment complies to the profile defined within the agreed SLA. The time required for this installation is difficult to predict, since it might be necessary to download installation packets from remote systems. Hence, in practice a fix amount of time is assumed for this installation procedure (e. g. five minutes), more than large enough to cover all typical node initializations. For parallel applications, the initialization effort of a new node is nearly static regardless of the number of nodes contained in the partition assigned for executing the parallel application.

In the context of an SLA-aware RMS offering an agreed service quality level on storage and network, the RMS also needs to initialize these domains at this point. Regarding the network the user is able to request properties like minimum bandwidth or exclusive medium access. For this the RMS has to interface the network management, initializing it accordingly. This task is not time consuming and can be performed within the fixed time interval for general initialization. Likewise the storage management has to initialize storage capacity according to the terms of the agreed SLA.

The time for initialization of job i can be defined as:

$$t_i^{initnew} = t^{nodeinit} + t_s^{stinit} + t^{stmount} \tag{3.1}$$

with $t^{nodeinit}$ as fixed time for node initialization, t_s^{stinit} as time for storage initialization with size s, P_i as partition of job i, and $t^{stmount}$ as time for mounting the storage. Depending on the actual software system used for providing storage services, $t^{stmount}$ may depend on the number of nodes in the partition of job i.

Both solutions resulted in an effort for rollbacking the storage linear to the size of their storage partition. In case of NFS this rollbacked storage is automatically available on all compute nodes. Otherwise, if storage is provided on dedicated block device, it may be necessary that the block device holding the rollbacked storage is re-mounted to all compute nodes. If this re-mounting was successful, the computation should resume. This effort was linear to the amount of compute nodes.

In the case of a job restart, this initialization phase mainly consists of the same steps as in the case of a new job start. First the environment of the node in the node partition has to be initialized such that the job can be restarted according the specific checkpoint dataset profile. This profile may be part of the negotiated SLA. In contrast to the initialization of a new job, the storage container here does not have to be initialized and formatted. Instead, the storage checkpoint contained in the checkpoint dataset has to be rollbacked, i. e. restored to the storage partition or storage container. The time effort of this task is also linear to the size of the storage checkpoint. Concluding, the rollbacked storage container has to be mounted to the compute nodes.

Here the time for initialization of the restarted job i can be defined as:

$$t_i^{initrestart} = t^{nodeinit} + t_s^{strollback} + t^{stmount} \tag{3.2}$$

with $t_s^{strollback}$ as time for storage rollback with size s.

At the end of the job the partition of the cluster has to be released. In the post-runtime phase the RMS configures this partition back to normal operation. This implies the reconfiguration of the network subsystem, the release of storage containers, or the deinstallation of previously installed libraries and tools. This process takes the time t_i^{deinit}, not depending on parameters like number of nodes within the partition or size, size of data partition, or amount of memory within these nodes. Since the exact demand of time of all necessary operations is hard to predict, it should be roughly estimated, sufficiently grand to cover all potential operations.

3.4.2 Overhead caused by Checkpointing

The main instrument for providing fault tolerance is the generation of regular checkpoints. There are several checkpointing solutions available, working at application level, user level, or kernel level (refer to section 6.2). Within the scope of this work, the focus is on kernel level checkpointing only. Even if other solutions also provide checkpointing functionality in general, they do not fulfill the demand on application transparency.

Kernel level checkpointing solutions have in common that it is possible to checkpoint an application without the need of recompilation or relinking of this application. However, existing solutions differ in their functionality regarding prerequisites to support an application. These prerequisites are known to the administrator such that these demands can be handled in the negotiation process: if the user requests for fault tolerant service, the system adds these given constraints to negotiation process. By agreeing on such an SLA the user confirms that his application complies to the functionality restrictions of the particular checkpointing solution.

A further similarity of all checkpointing solutions which is of importance for the scheduling component is the fact that the completion of applications is delayed.

- Impact on performance of application: Some checkpointing solutions virtualize the entire system environment of a running application. Even if this virtualization is highly optimized and not compute intensive, it has impact on the overall performance of the compute node. Performance evaluations on an existing checkpoint solution revealed a slowdown of approximately 1 percent. This value may differ for other checkpointing solutions.

- Additional effort for checkpointing operation: The size of a checkpoint more or less equals the size of main memory consumed by the checkpointed application. In case of parallel applications, each node in the partition is checkpointed, thus increasing the total size of the checkpoint dataset. The compressibility of the checkpoint dataset is highly application dependent. Checkpoints of most tested applications were poorly compressible, not worth additional delay until resume of the application.

  The time for the pure checkpointing operation depends on the time required for dumping the memory allocated by the application on a node. In parallel applications this operation is executed on all nodes in parallel. However, here the particular checkpoint datasets have to be saved to a network file system, which may be a bottleneck. Some systems provide the option of first saving the checkpoint to a local volume, then copying it to a central network place at a later time. This way, the checkpointed

application can resume earlier. However, network and disk utilization may slowdown the application.

At negotiation time the resource management system does not have any knowledge about the actual future memory usage of the application. Hence, the system has to calculate using the worst case where the application consumes all available main memory. This parameter has been either specified by the requesting customer in the negotiation process, or the system has to calculate with the amount of actually available main memory on the compute nodes.

Tests performed on checkpointing performance revealed a linear relationship between consumed main memory and the checkpointing time for a single node. In addition, the checkpointing time of parallel application increased linear to the number of nodes used for parallel execution.

The factors for delay caused by performance degradation, per node checkpointing, and multi-node checkpointing are system specific. Parameters impacting this factor are manifold, e. g. processor performance, type and performance of network interconnect, or performance of network storage. The administrator has to test the impact of checkpointing on the deployed system, adjusting the parameters in RMS configuration.

Having these parameters set the new execution time of job i can be computed as:

$$\hat{t}_i = f^{slow} t_i \tag{3.3}$$

with f^{slow} as slowdown of job execution due to the checkpointing environment, t_i as original execution time of the application (as specified by the user).

In the case that the checkpoint system checkpoints a parallel application sequentially node by node, the time of a single checkpoint operation for job i can be defined as:

$$t_i^{cp} = |P_i| \, t_m^{cpnode} + t_s^{snapshot} \tag{3.4}$$

with t^{cpnode} as execution time of a checkpoint on a single compute node with m main memory, and $t_s^{snapshot}$ as time used for snapshoting the storage data of size s.

In the case of using MPI, this only covers the checkpointing of all running MPI instances of an MPI application. In fact, also the master MPI process (e. g. mpimon of Scali MPI Connect software) needs to be checkpointed. Since the size of this process is very small, the effort can be neglected compared to the effort of checkpointing large compute instances.

Depending on the selected MPI flavor and checkpointing system, the value of t_i^{cp} can be significantly smaller. IBM Metacluster generates a checkpoint of all

MPI instances in parallel, then copying in parallel the checkpoint information to a network filesystem location. This operation can be further accelerated by using the background transfer capabilities. Here, Metacluster checkpoints all MPI instances of an application in parallel, then directly resuming the execution of the application. The transfer of the checkpoint datasets of the MPI instances is then performed while the application is already running again. This way, the checkpoint of a parallel application running on multiple nodes converges the checkpoint effort for a single node application.

However, this performance improvement has two negative side effects. Firstly, the background transport of the checkpoint datasets consumes processor performance as well as network bandwidth. This may impact the performance of the running application, increasing the slowdown factor caused by the checkpointing environment or impacting the comparability of the compute results. Secondly, the determination of data consistency is significantly more difficult for the RMS, since the checkpoint subsystem returns the state successful directly after all MPI instances have been checkpointed. This way, the RMS is not directly informed if failures during the transfer of the checkpoint parts occur (e. g. due to insufficient disk capacity for storing all checkpoint parts, so that the checkpoint dataset does not contain the checkpoints of all MPI instances). In this case the RMS has to manually validate the old checkpoint dataset before generating a new one, respectively before restarting the application in case of a resource outage.

3.4.3 Overhead caused by Migration

In the case of resource outages, a fault tolerant resource management system should restart the job from the latest checkpointed state. For this, the checkpoint dataset has to be transferred to the particular target host(s).

As long as fault tolerance is provided on the intra-cluster scope, i. e. the cluster may only use its internal resources for resuming the job, the costs of dataset migration can be omitted. This is due to the fact that cluster systems are usually operating with a shared network filesystem, so that a checkpoint dataset is easily available on each cluster node.

If migrating the checkpoint dataset over the border of a cluster system, there usually is no network filesystem shared with the source cluster system. Therefore the checkpoint dataset has to be transferred over the network, using either standard data transfer mechanisms (e. g. file transfer protocol (FTP) or secure data copy (SCP)) or other network transport mechanisms (e. g. data transmission protocols of Grid middleware). Since regular public network connections are used in this context, the bandwidth between source and target cluster system is the limiting factor.

The time required for the migration process can be defined as:

$$t_i^{migr} = \frac{s_i^{cp}}{f_{A,B}^{bw}} \tag{3.5}$$

with s_i^{cp} as size of checkpoint dataset of job i to be migrated and $f_{A,B}^{bw}$ as bandwidth between cluster systems A and B.

Due to the general lack of network bandwidth reservation mechanisms in regular networks, the actual bandwidth available at migration time is hard to predict. Especially if migrating over wide area network connections (e. g. by migrating over the Grid) the bandwidth is close to unpredictable. First this is due to the fact that source and target clusters may be even on different continents, having slow, instable, or congested network segments between them. Secondly, the target resource is unknown at job start, because the RMS locates suitable target resources within the Grid only in case of resource outages.

The measurement of network connections within the Grid as well as the reservation of network bandwidth within the Grid are topic of ongoing research (e. g. within the Network Measurements Working Group (NM-WG) of the GGF [75]). Having such mechanisms available, network bandwidth as well as resources for migration could be reserved at start time, such that this would result in an increased service quality level for the running application.

Without such mechanisms the bandwidth on communication interconnects has to be estimated. It has to be differentiated between local area networks (i. e. connecting other clusters within the local domain) and wide are networks (i. e. migration to remote Grid resources).

3.4.4 Overhead caused by Restart

By providing fault tolerance service to a running application, the resource management system creates checkpoints in well defined periodic intervals. In case of a resource outage, the latest checkpoint dataset is used for restarting the application.

Such a restart implies that all computational results between the moment of the resource outage and the latest checkpoint are lost. At worst case, the resource outage occurs shortly before the new checkpoint or even during the generation of a new checkpoint, so that an entire computation interval is lost and has to be repeated.

The length of a compute interval of job i can be defined as:

$$t_i^{compint} = \frac{\hat{t}_i}{n_i^{cp} + 1} \tag{3.6}$$

with n_i^{cp} as the number of planned checkpoints for job i. If n_i^{cp} equals to zero (i. e. no checkpoints are planned) the entire application has to be restarted from scratch.

3.4.5 Determining the Minimum Slot

If the RMS analyzes the schedule to find free slots for computing the schedule, it does not scan for slots which are large enough to compute the entire job including possible restarts. The RMS assumes that the job execution will be successful, taking the resource outage as an unlikely event. Therefore it only scans for slots which are large enough to compute the job and to execute all regular fault tolerance mechanisms:

$$t_i^{minslot} = t_i^{initnew} + \hat{t}_i + n_i^{cp} t_i^{cp} + t_i^{deinit} \tag{3.7}$$

with n_i^{cp} as the number of planned checkpoints on this job i.

3.4.6 Determining the Checkpoint Frequency

The definitions of $t_i^{compint}$ and $t_i^{minslot}$ underlined the importance of the checkpointing frequency. The selection is a trade-off between overhead in checkpointing and overhead in restarting. If no resource outages occur, a long checkpointing interval is beneficial since resources are not used for generating checkpoints, but for valuable computations. Moreover the satisfaction of the customer is higher since he gets his results as early as possible. In contrast, in case of failures a small interval prevents repeating large parts of the computation.

In case of deadline bound jobs, the resource management system has to ensure an appropriate time buffer for allowing the system to repeat the compute interval which was affected by the resource outage. Reserving buffer space for one restart implies that the computation may be interrupted once during the application runtime. The number of failing nodes during this outage does not matter, since the time for repeating the computation of the last interval does not depend on the number of failing nodes.

If the negotiation included the handling of multiple resource outages, the buffer space available for restart has to be increased accordingly. In case the number of restarts has not been specified, the system has to use a default value coming from fixed configuration or long-term experience.

Concluding, the total overhead for restart results in:

$$t_i^{jobmax} = \hat{t}_i + n_i^{cp} t_i^{cp} + n_i^{romax}(t_i^{migr} + t_i^{compint}) \tag{3.8}$$

with n_i^{romax} as the maximum number of covered resource outage events allowed for job i. Here, we assume the worst-case, where the resource outage

happened just in the moment of the next checkpoint execution, making the re-computation of the entire interval $t_i^{compint}$ necessary.

The determination of the checkpoint frequency depends on multiple factors:

- Minimum checkpoint frequency f_i^{cpmin} of job i is defined by the time span available from start date until deadline as well as the maximum number of covered resource outage events. The frequency has to be chosen high enough to keep the $t_i^{compint}$ term small enough to match with the given deadline. If the system would decrease the number beyond that point, the worst case finishing time (i. e. assuming that the maximum number of covered resource outages take effect) would exceed the given deadline.

- Maximum checkpoint frequency f_i^{cpmax} of job i is defined analogously. Even if $t_i^{compint}$ is reduced by increasing the checkpoint frequency, the term $n_i^{cp} t_i^{cp}$ increases. The maximum frequency is the point where the time span available from start date until deadline as well as the maximum number of covered resource outage events.

However, the checkpointing frequency does not solely depend on the time until the given deadline. Also the length of the free slot within the schedule has to be regarded.

On scheduling time, the RMS scans the schedule for a free timeslot for the job in question. This slot has to be large enough so that the job can be checkpointed at least often enough, so that it can finish before the deadline even in worst case of failures:

$$f_i^{cpmin} \leqslant f_i^{cpact} \tag{3.9}$$

with f_i^{cpact} as the actually chosen checkpointing frequency at runtime of job i.

Analogously there is no point in increasing the checkpoint frequency beyond the point, where the slot is no longer large enough to compute it before the deadline even without resource outages occurring, which is denoted as $f_i^{cpslotmax}$. These constraint can be defined as

$$f_i^{cpact} \leqslant \min(f_i^{cpmax}, f_i^{cpslotmax}) \tag{3.10}$$

These constraints combine to:

$$f_i^{cpmin} \leqslant f_i^{cpact} \leqslant \min(f_i^{cpmax}, f_i^{cpslotmax}) \tag{3.11}$$

Since the provider is interested in a high utilization of his cluster machines, he will presumably tend to decrease the checkpointing frequency as much as possible. This way, a smallest possible fraction of compute power is used for fault

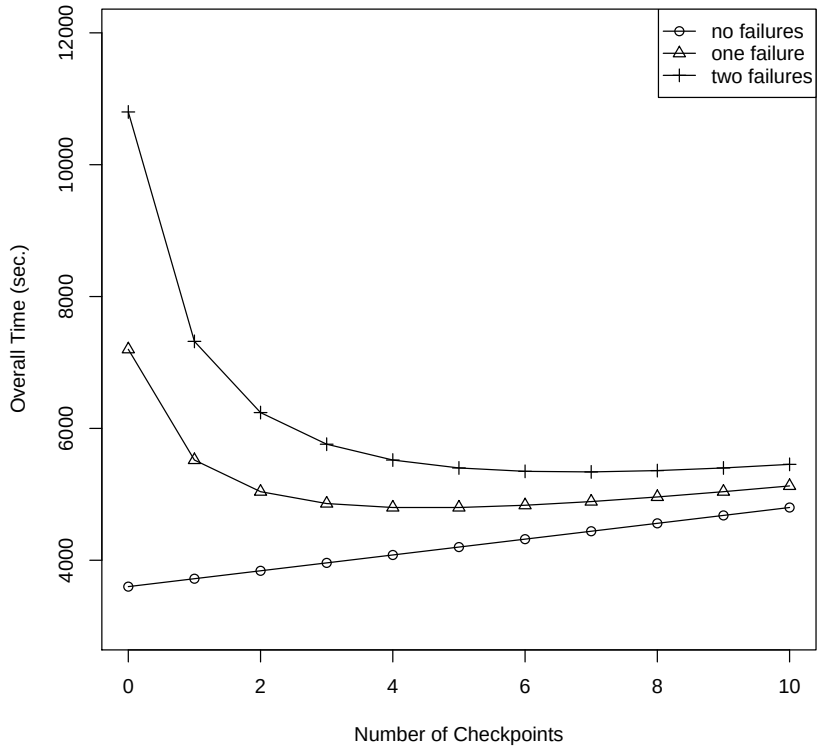

Figure 3.2: Impact of Checkpoint Frequency on Runtime

tolerance purposes, resulting in more available space for paid job computation. Depending on the different levels of guarantees described in section 3.2.2, the provider will limit his efforts as much as possible.

By negotiating on a given number of resource failures that should be handled, the resource management system is able to decide on a minimum checkpointing frequency. When negotiating on full service, the resource owner is able to configure the maximum number of resource failures he wants to cover in his calculation, thus impacting the selection of checkpointing frequency here.

The requesting customer is also able to negotiate on a given checkpointing frequency, despite of the chosen level of guarantee. According to the system utilization known at negotiation time, the resource management can scan the schedule looking for free slots able to perform the requested checkpointing frequency. If such a slot can be found, the negotiation may be accepted. If no such slot is available or the selected checkpointing frequency violates the listed constraints, the negotiation request has to be rejected.

The impact of the chosen checkpoint frequency on the runtime of a job ac-

cording to equation 3.8 is depicted in figure 3.2. 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 for $n_i^{cp} = 0$, 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, 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 power in case of resource outages, 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.

The minimum can be determined by differentiating equation 3.8 on the number of checkpoints, omitting the time required for job migration, since we assume a local restart. We have $\hat{t}_i$ as runtime of the job, t_i^{cp} as time required for a single checkpointing operation, and n_i^{romax} as the maximum number of expected resource outages.

We know: $\hat{t}_i > t_i^{cp} > 0$. Since $n_i^{romax} = 0$ is trivial, we furthermore assume $n_i^{romax} > 0$.

$$f(x) = \hat{t}_i + x t_i^{cp} + \frac{n_i^{romax} \hat{t}_i}{x+1} \tag{3.12}$$

$$f'(x) = t_i^{cp} - \frac{n_i^{romax} \hat{t}_i}{(x+1)^2} \tag{3.13}$$

$$f''(x) = n_i^{romax} \hat{t}_i \frac{2x+2}{(x+1)^4} \tag{3.14}$$

Calculating the root of the first differentiation as mandatory prerequisite for local extrema results in:

$$f'(x) = 0 \tag{3.15}$$

$$\Rightarrow \quad x_{1/2} = -1 \pm \sqrt{n_i^{romax} \hat{t}_i / t_i^{cp}} \tag{3.16}$$

Taking the example figure above, setting $\hat{t}_i = 3600$, $n_i^{romax} = 2$, and $t_i^{cp} = 120$, this computes to: $x_1 = -1 + \sqrt{2 \cdot 3600/120} = -1 + \sqrt{60} \approx 6,745967$ und $x_2 = -1 - \sqrt{60} = -8,745967$.

For ensuring that x_1 is a local minimum, we finally calculate the second differentiations.

$$f''(x_1) = n_i^{romax}\hat{t}_i \frac{-2 + 2\sqrt{n_i^{romax}\hat{t}_i/t_i^{cp}}}{(n_i^{romax}\hat{t}_i/t_i^{cp})^2} \tag{3.17}$$

$$f''(x_2) = n_i^{romax}\hat{t}_i \frac{-2 - 2\sqrt{n_i^{romax}\hat{t}_i/t_i^{cp}}}{(n_i^{romax}\hat{t}_i/t_i^{cp})^2} \tag{3.18}$$

Due to $\hat{t}_i > t_i^{cp}$ it follows $n_i^{romax}\hat{t}_i/t_i^{cp} > 1$. Therefore $f''(x_1) > 0$ und $f''(x_2) < 0$, resulting in $x_1 = -1 + \sqrt{n_i^{romax}\hat{t}_i/t_i^{cp}}$ as local minimum, and x_2 as local maximum.

Obviously x_2 is not a valid solution, because it is impossible to execute a negative number of checkpoints. Therefore the rounded up value of x_1 represents the optimal number of checkpoints for this example: $\lceil x_1 \rceil = 7$

Calculating the total runtime when executing seven checkpoints computes to: $f(7) = \hat{t}_i + xt_i^{cp} + \frac{n_i^{romax}\hat{t}_i}{x+1} = 3600 + 7 \cdot 120 + \frac{2 \cdot 3600}{8} = 5340$.

If generating seven checkpoints, the job is running 5340 seconds (1 hour and 29 minutes) instead of only one hour, but being interrupted two times by resource outages.

3.5 Fault Tolerance with intra-cluster scope

Fault tolerance is a crucial building block for realizing an SLA-aware resource management system, which does not only negotiate on SLAs, but is also able to adhere to agreed deadline in case of resource outages. Beside integrating fault tolerance mechanisms into the RMS, also the system management (i. e. the system scheduler) has to support these mechanisms.

In this section the impact of providing fault tolerance to the system scheduler will be outlined. The system will only be able to use its internal resources to compensate resource failures. This will be enhanced in the following by migration techniques to remote resource, e. g. resources on different clusters within the same administrative domain or even resources somewhere in the Grid.

The scheduler of a resource management system is invoked each time when scheduling decision has to be taken or an event occurs, which impacts one of the jobs within the schedule.

- New request: If a new request enters the system the scheduling component is queried. It has to decide if the new job can be accepted or has to be

rejected. This decision has to be taken in the context of the current schedule as well as the current system condition.

- Job finishes: After successful completion, a job terminates successfully. In this case the resource management has to start the post-runtime procedures and finally free the allocated resources for new jobs. By using planning based resource management, each user has to estimate the runtime of his jobs. If the job terminates before the estimates runtime (i. e. the user overestimated the runtime), the resources are released earlier than expected. This results in unexpected gaps within the schedule. The RMS may assign these gaps to other waiting jobs.

- Exception handling: The RMS has to ensure the adherence with given SLAs at runtime. In case of resource outages or non-compliant performance characteristics, the RMS first has to identify the jobs affected by this exception. Then the scheduler has to decide on using its mechanisms of fault tolerance to ensure the SLA-compliance. In this context, the RMS may decide to migrate the job to remote resources.

Data: R as set of all running jobs
Data: S as set of all accepted SLA jobs
Data: P as set of all problematic jobs, e. g. affected by outages or non-SLA-compliant performance characteristics, ordered by type, value descending
Data: B as set of all best-effort jobs
Data: N as set of all new jobs requests, ordered by value descending
Result: Sched^{New} as new schedule
Result: P as set of all problematic jobs
Result: N as set of all unaccepted new job requests

Figure 3.3: Parameters for Scheduling Algorithm

Beside these events, there are also two further major events within the RMS. However, these events do not impact the scheduling.

- Job starts: A planning based resource management system assigns a start time to all jobs within the system. If such a start time is reached, the RMS initiates the pre-runtime phase, following by the execution of the application on the nodes of the assigned cluster partition.

- Runtime exceeds: If the user underestimated the runtime of his job, the RMS will terminate the job at a given point of time. Depending on the

system policy, the job is either terminated or suspended. The system may also decide on runtime prolongation of the job, if this can be realized in the situation of the current system schedule.

3.5.1 Basic SLA-aware Scheduling

This subsection focuses on realizing SLA-aware scheduling. The parameters of the algorithm are depicted in Fig. 3.3.

3.5.1.1 Input Parameters

This algorithm has the following sets as input parameters:

- Set of all running jobs (R): This first set of jobs contains all jobs which are currently running on the machine. These jobs can be both SLA- and non-SLA-bound.

- Set of all SLA-bound jobs (S): This set encompasses all jobs, where the RMS agreed on some kind of SLA. This can be either the agreement to provided resources at a specific time (i. e. reservations) or to complete the job until a given time (i. e. deadline). Additionally SLAs may contain statements regarding performance characteristics of the resources. Jobs in S have been agreed, but are not running yet. Therefore:

$$\mathsf{R} \cap \mathsf{S} = \emptyset$$

- Set of all best-effort jobs (B): The RMS does not only execute jobs bound to an SLA, it also executes standard best-effort jobs. In contrast to SLA-bound jobs, the RMS did not agree on any guarantees for execution of these jobs. Therefore these jobs may be aborted or delayed arbitrarily. Similar to S, jobs in B have been accepted, but are not running. Therefore:

$$\mathsf{R} \cap \mathsf{B} = \emptyset$$

- Set of all problematic jobs (P): In case of resource outages or other unforeseen problems within the cluster, this may affect the fulfillment of jobs scheduled to execute on these resources. These jobs are elements of P. If a SLA-violation of a job in P is imminent, the RMS has to react accordingly. P may contain jobs from R, S, and B, so that

$$\mathsf{P} \subseteq \mathsf{R} \cup \mathsf{S} \cup \mathsf{B}$$

- Set of all new job requests (N): This set holds all requests for new jobs which have been submitted to the system. These requests can be both SLA bound and best-effort. The RMS tries to integrate the new jobs into the schedule. In case of SLA-bound jobs it has to verify if it is possible to fulfill all requirements of the SLA. If the RMS succeeds in this, it removes the job from this set.

The value of a job does not necessarily correspond to the price that the customer has to pay in case of SLA fulfillment or the penalty fee that the provider has to bay in case of SLA-violation. The value may also be determined by internal policy decisions of the resource owner, who may prioritize on targeting specific customer segments (i. e. jobs from VIP customers would be of higher value than jobs from normal customers) or utilization of the machine (e. g. value of job depends to the layout of the job, perhaps preferring long running sequential jobs, or massively parallel jobs).

It is essential that the elements of P are sorted in following order: R first, S second, B third. The value has to be used as secondary sorting key. This way the scheduler first tries to solve the problems of already running jobs, then taking all remaining resources for handling the problems of jobs from set S.

Each of these sets is sorted by the value of the job for the resource operator. Sorting the subsets of P following the value criterion guarantees that available resources are first used for handling the problems of valuable jobs.

Also the set N containing all new job requests has to be sorted according to their value. Similar the jobs in P, this value does not necessarily represent the price for SLA-fulfillment, but can represent an arbitrary objective defined in the internal system policies.

The scheduler is working on these input sets and tries to generate a new valid schedule for the cluster machine. These input parameters are identical for all following scheduling algorithms.

A job can only be element of one of the sets R, S, B, or N, so that:

$$\mathsf{R} \cap \mathsf{S} \cap \mathsf{B} \cap \mathsf{N} = \emptyset$$

Additionally, jobs can be elements of set P. Jobs in N are never elements of P, since these jobs are only in state of negotiation (ref. to Table 3.1). Therefore:

$$\mathsf{N} \cap \mathsf{P} = \emptyset$$

3.5.1.2 Output Parameters

Beside these input parameters, the scheduling algorithms return the result of the scheduling operation using the following return sets:

```
1  begin
2      Sched^New = ∅
3      S^Res = {j ∈ S|j is reservation}
4      S^¬Res = S \ S^Res
5      N^SLA,Res = {j ∈ N|j is bound to SLA, j is reservation}
6      N^SLA,¬Res = {j ∈ N|j is bound to SLA, j is not reservation}
7      N^¬SLA = N \ (N^SLA,Res ∪ N^SLA,¬Res)
8      P^horizon = {j ∈ P|j in problem horizon}
9      forall j in (R ∪ S^Res) \ P^horizon do
10         insert j into schedule Sched^New
11     forall j in P^horizon do
12         if j is SLA-bound then
13             if not conflict(j,Sched^New,S^¬Res \ P^horizon) then
14                 insert j into schedule Sched^New
15                 remove j from P
16     forall j in N^SLA,Res do
17         if not conflict(j,Sched^New,S^¬Res) then
18             insert j into schedule Sched^New
19             remove j from N
20     forall j in S^¬Res \ P^horizon do
21         insert j into schedule Sched^New
22     forall j in N^SLA,¬Res do
23         if not conflict(j,Sched^New) then
24             insert j into schedule Sched^New
25             remove j from N
26     forall j in B ∪ N^¬SLA do
27         insert j into schedule Sched^New
28         remove j from N
29 end
```

Figure 3.4: Algorithm: Basic SLA Scheduling

- The new system schedule ($\mathtt{Sched}^{New}$): The main task of each scheduling algorithm is to assign available system resources to waiting jobs. In the case of a planning based RMS, the scheduler assigns start times to all waiting jobs. However, in the case of non-reservations, these start times may be subject to change.

- Problematic Jobs (P): The scheduler tried to care about problematic jobs at runtime. Now, this set contains all the jobs where the scheduler was not able or did not try to find such a solution.

  A schedule may contain thousands of jobs, utilizing the machine for weeks or months. Within this time frame it is most likely that a failed node will be repaired and reintegrated into the cluster system. Therefore it is not necessary that the scheduler handles all jobs in P. Instead, the scheduler only works on jobs within a given horizon. This horizon has to be defined by the resource operator and should default to the regular time span that is required to fix problems on cluster nodes.

  – If the affected job is not yet in the problem horizon of the scheduler, it skipped solving the problem (e. g. resource failed now, but job is about to start in three weeks)
  – If the affected job is in the problem horizon, but the scheduler was unable to solve the problem (e. g. failing resource, but no spare resource available for resuming the job).

  Jobs remaining in P may cause an SLA-violation, if the scheduler is not able to solve the problem until the actual start of the job. If the scheduler decided on displacing an other job for executing a job in P, this displaced job is in the result set P of the scheduler. Hence, it is possible that P contains other (and even more) jobs as at start of the scheduler.

- The set of all unaccepted new job requests (N): The scheduler tries to find resources for all new job requests. In the case of SLA-bound jobs, the scheduler may decide to reject such a new job request, since it was unable to find system resources so that the adherence of the SLA could be assured. These rejected job requests are returned in the return set N.

3.5.1.3 Initialization

A basic algorithm for scheduling jobs with SLAs is depicted in Fig. 3.4. This scheduler is utilizing the system fault tolerance capabilities only for resuming SLA-bound jobs in case of exceptional situations.

In a first step this algorithm initializes the result schedule ($\mathtt{Sched}^{New} = \emptyset$) and generates subsets of the input sets S and N. This is necessary, since the

algorithm operates different on the subsets of these sets. It is presumed that the order of the elements in the sets is preserved in the generated subsets. The subset $\mathsf{P}^{\mathsf{horizon}}$ defines which problematic jobs have to be handled at this point of time. The subset will be described in the following.

At this point of time the schedule is empty. The scheduler therefore is able to take the entire (remaining) machine for placing the jobs.

3.5.1.4 Handling of Running and SLA-Reservation Jobs

Currently running jobs have the highest priority, because these jobs are currently consuming compute power. If the scheduler decides on not continuing the execution, these jobs are either aborted or the system has to use its mechanisms on checkpointing, so that the job can be resumed at a later point of time.

$\wedge$	$\notin \mathsf{P}$	$\in \mathsf{P}$	running	SLA
$\in \mathsf{R}$	no problems	if SLA-bound, problem on SLA-compliance, but running	yes	both
$\in \mathsf{S}$	no problems	future SLA-compliant execution currently not ensured	no	yes
$\in \mathsf{B}$	no problems	assigned resources for execution currently not available	no	no
$\in \mathsf{N}$	-	-	no	-

Table 3.1: Sets in Scheduling Algorithm

The same holds valid for all SLA-bound jobs connected to fixed reservations. It is vital that these reservations keep stick to their original places in the schedule, because these jobs may be part of a higher level workflow. If the scheduler would decide on relocating these reservations, this may endanger the success of the overall workflow. Therefore the user negotiated on this very time span, so that the scheduler should try to provide resources as agreed.

If a job in R or $\mathsf{S}^{\mathsf{Res}}$ is element of P, this implies that there is some kind of problem regarding the fulfillment of the SLA.

- In case of jobs in R, this implies that this problem does not affect the actual execution of the job (since the job is still running), but the execution violates terms of the SLA. In the case that a running job is affected by a resource outage that prevents further execution, it is removed from R and added to S or B (depending if it is SLA-bound or not) and P.

- If the job is in $\mathsf{S}^{\mathsf{Res}}$, the job is currently not running, but its future SLA-compliant execution is endangered.

In both cases, the job is not directly added to the new schedule, but subject to the following problem handling procedure.

The first action on the result schedule $\mathtt{Sched}^{\mathrm{New}}$ is to insert all currently running jobs as well as all scheduled SLA-bound reservation jobs. For this, the scheduler takes all jobs in the set R and $\mathsf{S}^{\mathsf{Res}}$ and adds them to $\mathtt{Sched}^{\mathrm{New}}$, if the job is not element of P. This step is essential, since it ensures the completion of all running jobs as formerly planned and agreed within the SLA. The jobs of R and $\mathsf{S}^{\mathsf{Res}}$ in P are not directly added to the schedule, since the RMS has to react according to the problem occurred.

3.5.1.5 Handling of Problematic Jobs

In the second phase the scheduler has to work on this set P. Due to the described sorting of P this set first holds elements of R, followed by elements from S. Each of these subsets is sorted by the value of the job, e. g. represented by the price for SLA-fulfillment that the customer has to pay. The scheduler now works on the set P element by element.

If the cluster system is affected by a resource outage, this event will be propagated to the system scheduler. The scheduler will in turn mark the nodes affected by this resource outage as defect. The administrator may specify that these resources will not be used within following negotiations (i. e. the capacity of the cluster system is reduced in the succeeding negotiation processes). However, the cluster system may already have agreed on SLAs, presuming the availability of all system resources.

As described, a planning based resource management system builds up an entire system schedule, containing start time and resources for all jobs. Hence, the scheduler is aware of the impact that a resource outage has on the total system environment. Since every job is directly assigned to actual resources, the system knows which jobs are affected by the resource outage. All these jobs are automatically added to P.

The scheduler will then only handle those jobs in P whose runtime is within the problem horizon. For all other jobs the scheduler assumes that the problem will be solved and the resource is available again if it comes to job execution. If the problem can not be solved within the horizon time frame, the administrator has the option to extend the horizon arbitrarily (e. g. the outage of a fan caused damage on a motherboard, which first has to be ordered, so that the node is unavailable for one week).

Depending on their membership in P or S and their particular value, all jobs within the horizon are now handled by the scheduler. It tries to find resources in

the new system schedule Sched^{New}. Since running jobs as well as SLA-bound jobs representing reservations have already been placed to Sched^{New}, the newly added job from P does not impact any time critical job.

Before placing the job from P to the new schedule Sched^{New}, the scheduler checks for conflicts with requirements of the SLA of this job. Only if all requirements are fulfilled (e. g. deadlines specified within the SLA can be met), the job may be added to Sched^{New}.

In a second validation the scheduler checks if the newly added job impacts the placement of jobs in $\mathsf{S}^{\neg Res}$ to the schedule. This placement will be the next step in this algorithm, directly following this handling of problematic jobs. The RMS did agree on SLAs for all these jobs in $\mathsf{S}^{\neg Res}$. It has to be avoided that the handling of problematic jobs affected by an outage again affects other jobs. Therefore the new job may only be placed to the new schedule Sched^{New}, if it does not conflict with any other agreed SLA-job from $\mathsf{S}^{\neg Res}$. In this context the system has to verify if deadlines of jobs in $\mathsf{S}^{\neg Res}$ can still be reached after placement of the new job in Sched^{New}, or if specific hardware resources are still available.

Only in the case where the placement of the new job from P in the new schedule Sched^{New} neither conflicts with its own SLA, nor with any SLA assigned to jobs in $\mathsf{S}^{\neg Res}$, it may be finally added to the new schedule. In this case, the job is removed from P, because it is no longer operating on resources which are affected by the resource outage. In the other case, the job remains in P and the scheduler continues with the next job found in P.

All jobs remaining in P after this step are critical, because the violation of their particular SLAs is imminent. For the resource owner this implies that he is in danger of penalty fees agreed within these SLAs.

3.5.1.6 Handling of SLA-Requests on Reservations

The set N encompasses all new job requests to the resource management system. N may contain both SLA-bound jobs and best-effort jobs. In this step of the algorithm, the scheduler works on all SLA-bound job requests requesting reservations, which are elements of the subset $\mathsf{N}^{SLA,Res}$. Since SLA-requests on reservations have a smaller degree of freedom while placing in the schedule, these jobs need to be handled by the algorithm before other job requests [39]. According to the order of elements in N, the subset is ordered by the value of the job.

At this point of time the new schedule Sched^{New} holds all running jobs (from set R), all agreed SLA-jobs requesting reservations (from set S^{Res}) and all recovered problematic jobs from P. The remaining space in the schedule now is used for adding SLA-requests on reservations from $\mathsf{N}^{SLA,Res}$.

The procedure of this step is similar on the previous step on handling prob-

lematic jobs. Again, the set of requests is iterated request by request. Also the insertion process is a twin step process:

- Conflicts with SLA of job: Before the job can be placed to the new system schedule $\mathtt{Sched}^{New}$ it has to be verified that all demands of the SLA-request can be fulfilled. In particular the reservation has to be realized in the context of the new schedule.
- Conflicts with jobs from $S^{\neg Res}$: Presuming that all demands of the new SLA-request can be fulfilled, it still has to be possible to realize all agreed SLA-jobs in $S^{\neg Res}$. If the addition of the new SLA-request to $\mathtt{Sched}^{New}$ violates the SLA-compliance of any job in $S^{\neg Res}$, the request may not be accepted.

If both conflict checks for the new SLA-request are negative i. e. no conflicts detected), the new job may be accepted. In this case the scheduler removes the job request from $N^{SLA,Res}$ and adds it to the new schedule. In the other case, the SLA request remains in $N^{SLA,Res}$ and the algorithm continues with the next SLA-request in $N^{SLA,Res}$. All requests remaining in $N^{SLA,Res}$ after this step have to be rejected.

3.5.1.7 Handling of SLA-Non-Reservation Jobs

In the previous steps of the algorithm the new schedule $\mathtt{Sched}^{New}$ has been filled with running jobs, SLA-jobs requesting reservations, recovered problematic jobs, and SLA-requests on reservations.

While adding problematic and new jobs it has been checked that these jobs do not interfere with already agreed SLA-bound jobs, which do not specify reservations. Therefore we can presume that the schedule holds sufficient capacity to realize all jobs remaining. Problematic jobs within the problem horizon $P^{horizon}$ do not have to be regarded here, since they have been already inserted during the second step of the algorithm.

This step now inserts all SLA-jobs of $S^{\neg Res} \setminus P^{horizon}$ to the new system schedule $\mathtt{Sched}^{New}$.

3.5.1.8 Handling of regular SLA-Requests

This step of the algorithm completes the addition of new SLA-jobs to the system scheduler. Again the jobs within $N^{SLA,\neg Res}$ are sorted according to their value, so that this algorithm first tries to add the most valuable task to the new system schedule $\mathtt{Sched}^{New}$.

In contrast to the step on handling SLA-requests on reservations, this step does not need to care on dependencies with already agreed jobs. This step can

analyze the current system situation and try to add as much new SLA-requests as possible.

At each of these iteration the scheduler checks if the new schedule allows an SLA-compliant placing of the new job. In that case the job can be accepted, so that the job is removed from N and added to the schedule. SLA-requests remaining in $\mathsf{N}^{\mathsf{SLA},\neg\mathsf{Res}}$ after this step have to be rejected.

3.5.1.9 Handling of existing and requested Best-Effort Jobs

If accepting best-effort jobs, the resource management system does not promise on providing any guarantees or service quality levels. In fact, the user is only provided with resources that are not used by other SLA-bound jobs. Moreover, the system does not use its mechanisms on fault tolerance to minimize the impact of resource outages to this kind of jobs. In case of such outages, the best-effort job is either cancelled or restarted from the very beginning.

Therefore this kind of jobs is added to the new system schedule in the last step of the algorithm. Here, the scheduler tries to utilize free gaps in the schedule to place these jobs. If it can not find matching gaps, the best-effort jobs are placed at the end of the system schedule.

Since a best-effort job can not put any demands on the RMS (e. g. to complete the job until a given deadline), the scheduler always has the option to put the best-effort job at the end of the schedule. Hence, requests on new best-effort jobs are never rejected.

Within the set of the best-effort jobs the scheduler follows the first-come-first-served (FCFS) policy, so that jobs may be delayed by other SLA jobs, but not by other best-effort jobs.

3.5.2 SLA-aware Scheduling with Job Suspension

The focal goal of an SLA-aware resource management system is the adherence with all agreed SLAs. Due to the penalty fee contained in each SLA, resource outages are potential business risks for the provider. Hence, the resource management system has to use its mechanisms in such a exceptional situation as good as possible to minimize the financial impact for the provider.

With the presented algorithm in section 3.5.1 SLA-aware scheduling is introduced into resource management. The RMS is able to cope with exceptional situation since the scheduler uses available spare resources for handling problematic jobs within the problem horizon according to agreed SLAs.

It becomes apparent that the amount of available spare resources limits the options of the resource management system to compensate outages and exceptional situations. Given that sufficient suitable spare resources are available, the system is able to resume all affected jobs from the latest checkpointed state,

thus guaranteeing the adherence with the agreed SLAs of these jobs. However, resources in a cluster system are limited and usually already blocked by other waiting jobs or jobs which are currently in execution. Therefore spare resources are an exceptionally precious good in outage situations.

Introducing job suspension as means of system management allows to increase the number of available resources in such a situation. Here, the resource management system uses its capabilities on checkpointing and restart to stop jobs which are currently in execution, so that resources used by these jobs are free for resuming jobs affected by the resource outage.

If looking to the set of running jobs, one can distinguish between two different types of jobs:

- SLA-bound jobs: this job is bound to an SLA, where the RMS agreed on providing a specific service level. These jobs can be further classified into:
  - jobs bound to a fixed reservation
  - jobs not bound to any reservations, but other SLA-attributes (e. g. the adherence to a deadline or the provision of performance characteristics).
- best-effort jobs: this job is not bound to an SLA, so the RMS did not contractually agree on providing any services at any quality levels. In particular, the RMS does not have to ensure the adherence to any deadlines, nor the fulfillment of reservations, where resources have to be provided within a given fixed time frame.

These running jobs came to execution, since the scheduler assigned compute time according to its internal scheduling rules and policies. At the time of jobs start the scheduler assumed that the entire cluster is available, not anticipating the occurred resource outage. In the light of the current system situation it may have been better to postpone the execution of some of these jobs, so that these resources are available in this moment of fault compensation.

3.5.2.1 Suspension of Best-Effort Jobs

A first step in increasing the amount of spare resources is to use resources blocked by running best-effort jobs. Since these jobs are not bound to any SLAs, the RMS does not have to return any results at a specific time.

The easiest solution therefore is to cancel the execution of all best-effort jobs, if there are problematic jobs that need to be handled. This can be integrated by modifying the first step of the algorithm: instead of adding all elements of set R, it only adds those elements j of set R, where j is bound to an SLA (ref. Figure 3.5).

```
1 begin
2   forall j in R \ P^horizon do
3     if (B = ∅) or (j is bound to SLA) then
4       insert j into schedule Sched^New
5     else
6       add j to B
7   forall j in S^Res \ P^horizon do
8     insert j into schedule Sched^New
9 end
```

Figure 3.5: Modified Initial Step in Basic SLA Scheduling Algorithm

Since alls jobs in set B are ordered according to the FCFS model, the cancelled best-effort jobs would be put back into the common pool of best-effort jobs into their particular position:

- In case that backfilling mechanisms caused an out-of-order execution of this job due to a fitting gap within the schedule, the job has to be selected for backfilling again. Otherwise (i. e. if no matching gap exists in the schedule which allows an out-of-order execution), the job has to wait until its regular node assignment time according to the FCFS scheme.

- In case that the job was selected for execution due to its position in the FCFS scheme, the job remains at this position. Hence, it is first chosen among all best-effort jobs the next time a resource is available for execution of best-effort jobs.

After terminating the execution of all currently running best-effort jobs, the utilization of the machine is lower. Since best-effort jobs are added to the new schedule Sched^{New} only in the last step of the algorithm, the newly available space can be used for scheduling problematic jobs from set P.

Even if this approach improves the level of fault tolerance, it has two major disadvantages:

- Waste of computational power: Terminating already running jobs does not only increase the number of spare resources, but also voids all computational results of the terminated jobs. If thinking of long running applications, the amount of wasted computational power is significant. Moreover, the cluster has to repeat the voided computation of the terminated applications at a later point of time, thus blocking valuable resources.

- Lack of adequacy: Having problematic jobs in B at the start of the algorithm does not necessarily imply that these problems can not be solved without terminating best-effort jobs. Even if the number of spare resources is not sufficient for handling all problematic jobs, it may be sufficient to terminate only a fraction of the currently running best-effort jobs.

The first problem can be solved by utilizing the system's capabilities on checkpointing and restart. For this, each best-effort job is started by the resource management system as an SLA-bound job that should be regularly checkpointed. This way the system is able to generate a checkpoint of the running best-effort application if the scheduler decides on terminating it. The system saves the checkpoint dataset to the internal checkpoint repository. As soon as the best-effort job comes to execution again, it does not restart from scratch, but resumes its computation from this checkpointed state.

If adding the checkpointed and terminated job to the schedule, the system adjusts the remaining runtime of the job:

$$t_i^{remaining} = t_i^{remaining} - (T_i^{checkpoint} - T_i^{start})$$

The remaining runtime of the job results from the formerly remaining runtime of the job minus the already computed runtime, which is the time of the latest checkpoint $T_i^{checkpoint}$ minus the time of job start T_i^{start}.

At job start, the remaining jobtime $t_i^{remaining}$ was initialized by the runtime of the job in the checkpoint environment ($\hat{t}_i$). By subtracting the already computed time from the total compute time, the scheduler assigns a smaller segment to the rescheduled best-effort job. Hence, the job has a higher chance to fit into gaps within the schedule.

For the second problem, the scheduler has to evaluate the impact of a running best-effort job on the total schedule. Here it is not sufficient to determine the area in the schedule that is missing for successfully handling a problematic job, because an arbitrary number of jobs may be scheduled between the problematic job to be handled and the running best-effort job. Killing a best-effort job therefore would not automatically imply having corresponding spare resources to place the problematic job.

The problem can be solved by iteratively evaluating the impact of each best-effort job on the schedule. In this context we have to introduce the term of the area of a job. We define

$$a_i = |P_i| * \hat{t}_i$$

as the number of resources used by job i multiplied by the compute time. Analogously we define a_i^{remain} as the number of resources used multiplied by the remaining compute time of the job.

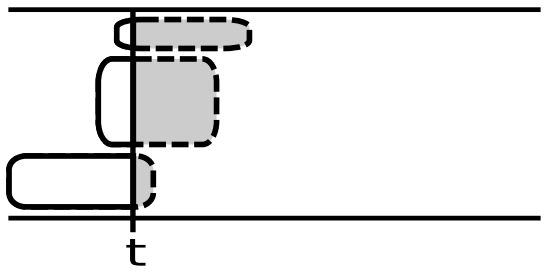

Figure 3.6: Best Effort Jobs in Schedule

Figure 3.6 depicts the impact of taking the remaining job area as selection criterion for job suspension. We have a starting situation here, where all running jobs are placed in the schedule. In this example, all running jobs are best-effort. If the system would select the bottom job, it would remove a large job from the system, but would not free significant amounts of resources, because most of the computation time is already passed. Only the grey colored part of the job would be available for other jobs.

It is not sufficient to match the area of jobs in P with the area of running best-effort jobs for identifying potential suspension candidates, because the algorithm of section 3.5.1 places jobs in P according to their value. This results in a higher probability for high value job in P to be placed than low value jobs. By removing best-effort jobs, the freed place in the schedule could be allocated by higher value jobs from P.

The idea of this algorithm is first remove jobs with a large remaining areas from the schedule. For this, it generates a copy of the current schedule (containing all running jobs, as depicted above) and start removing with the job having the largest remaining area. It continues removing the next largest, until 50 percent of the total remaining best-effort job area has been removed. Now it uses this copy for executing the scheduling algorithm presented in section 3.5.1. If this algorithm succeeds in placing all jobs from P, this implies that it is sufficient to remove 50 percent of the remaining best-effort area. In the other case, 50 percent is not sufficient.

In the second iteration the algorithm follows the idea of binary search, taking only 25 percent (if all jobs from P could be placed) or 75 percent of the total remaining best-effort area. The algorithm either terminates on identifying the sufficient area or by detecting that even stopping all running best-effort jobs is not sufficient for placing all jobs in P.

After this brief outline, the algorithm is now described in more detail. The scheduler first sorts the set of running best-effort jobs by their remaining area a_i^{remain}. It furthermore initializes two variables named $\mathtt{upperbound}$ and $\mathtt{lowerbound}$ to 100 and 0. The scheduler now generates a subset BE of the running best-effort jobs, containing the first jobs that have a total remaining

area of 50 percent (i. e. $\frac{\mathtt{lowerbound}+\mathtt{upperbound}}{2}$) of all running best-effort jobs. This subset is now removed from the set R and added to set B.

Having these sets modified, the previous SLA-scheduling algorithm is executed again. Depending on the result set P this algorithm may have the following result:

- $\mathsf{P} \neq \emptyset$: The scheduler is not able to place all problematic jobs with $\frac{\mathtt{lowerbound}+\mathtt{upperbound}}{2}$ percent of all remaining area of running best-effort jobs. Therefore set $\mathtt{lowerbound} = \frac{\mathtt{lowerbound}+\mathtt{upperbound}}{2}$.

- $\mathsf{P} = \emptyset$: The scheduler is able to place all problematic jobs, so that no problematic jobs remain. Even if the goal of SLA-compliance is achieved for all planned jobs, the scheduler might have terminated too many best-effort jobs. Therefore set $\mathtt{upperbound} = \frac{\mathtt{lowerbound}+\mathtt{upperbound}}{2}$.

With a new value for $\mathtt{upperbound}$ or $\mathtt{lowerbound}$ the scheduler again generates a subset BE or all running best-effort jobs and executes the SLA-scheduling algorithm.

This procedure terminates if $\mathtt{lowerbound} = \mathtt{upperbound}$. If the set P still is not empty, the scheduler is not able to handle all problematic jobs even if it terminates all running best-effort jobs. In the case that P is empty, the scheduler identified the set of running best-effort jobs that need to be terminated, so that all problematic jobs are handled and SLA-compliance is ensured.

3.5.2.2 Suspension of SLA-bound Jobs

If suspending all running best-effort jobs is not sufficient for compensating the SLA-related consequences of a resource outage, the system has to take further actions to increase its level of fault tolerance. The most obvious approach is to enhance the mechanism of suspending running best-effort jobs also to SLA-bound jobs.

As stated, these SLA-bound can be distinguished depending if they include a fixed reservation or not. In the first case, the resource management system must not suspend the job, because this may impact a higher level workflow. If the workflow engine has to ensure that tasks in different branches of a workflow execute in parallel, it is realized by means of reservations on local RMS. If the RMS would decide on suspending such a reservation job, this also impacts the tasks in the other branches, because the synchronicity is no longer ensured.

In contrast it is uncritical to suspend non-reservation jobs, e. g. demanding for a given deadline. Therefore the scheduler is able to apply the same mechanism of job suspension as in the case of best-effort jobs:

- Checkpoint: as soon as the scheduler decides on terminating a running SLA-bound job, it creates a checkpoint of this job and saves it to the internal checkpoint repository.
- Reschedule: the suspended SLA-bound job is then added to the new Schedule, such that all terms of the SLA are fulfilled, e. g. that the deadline is met. On adding the job to the schedule, it is only added using the remaining computation time.
- Resume: as soon as the job is executed again on a compute resource, the job is resumed from the latest checkpointed state.

The suspension of SLA-bound jobs is initiated in the case that suspension of best-effort jobs is not sufficient for guaranteeing the SLA adherence of all problematic jobs. The mechanism is similar, again using the remaining area of the job as deciding factor for job selection.

The scheduler first sets $\mathtt{lowerbound} = 0$ and $\mathtt{upperbound} = 100$ and generates a subset SJ encompassing $\frac{\mathtt{lowerbound}+\mathtt{upperbound}}{2}$ percent of the running SLA-bound jobs from R. This subset SJ is then removed from R.

It has to be ensured that the job will be executed in a way that the SLA is fulfilled. Therefore the scheduler adds the elements of SJ to the top of S, a set holding already agreed SLA-jobs. Thanks to their position in S, the newly added jobs are handled first by the scheduler. This way it is ensured that the suspended SLA-jobs come to execution first among all the agreed SLA jobs. Moreover the conflict detection mechanisms in the SLA algorithm ensure that their SLA-compliance is regarded while handling problematic jobs.

Having modified these sets, the SLA algorithm is now executed. According to the number of elements in the result set P, the $\mathtt{lowerbound}$ and $\mathtt{upperbound}$ variables are now adjusted, the set SJ regenerated and the algorithm executed again.

Corresponding to the suspension of best-effort jobs, this suspension procedure ends if $\mathtt{lowerbound} = \mathtt{upperbound}$. If the return set P is empty, the set of SLA-jobs has been identified that allows a successful handling of all problematic jobs. If P is not empty, even suspending SLA-bound jobs is not sufficient for handling all problematic jobs.

3.5.3 Partial Execution

Unfortunately users usually do not align their computational demands to the resource utilization of existing cluster machines. Therefore the resource management system of such a cluster machine receives uncoordinated resource requests, and assigns them to the available resources. In consequence to this lack of coordination the jobs do not fit optimal, so that the scheduler is forced to leave

gaps in the generated system schedule. These gaps have the effect that some compute resources of the cluster machine remain idle over the time span of the gap.

Scheduling techniques like backfilling have been introduced to deal with this problem. Here, the scheduler analyzes the gap and tries to find matching jobs somewhere in the schedule that would fit into this gap. If such a request is found, the scheduler moves the request to the particular gap. Backfilling is an important instrument in increasing the system utilization and numerous variations exist, each focusing on a specific administrative goal. However, backfilling has its limitations on improving the schedule quality if available requests do not match with existing gaps.

With checkpointing and restart the SLA-aware resource management system has a powerful instrument, which can be also applied to this problem domain. The basic idea is to use existing gaps in the schedule for partial job execution. In contrast to backfilling, this method does not depend that the runtime of potential jobs matches with the length of a gap. The only constraint is the number of nodes that the job requests.

3.5.3.1 Increasing Level of Fault Tolerance

Partial execution can also be used for increasing the level of fault tolerance. By suspending running best-effort and SLA-bound jobs, the resource management system did its best to handle all problematic jobs. If this is not yet sufficient, the system may decide to use the idle computational power of existing gaps to partially execute problematic jobs, thus trying to meet with SLAs.

Obviously partial execution is not applicable where fixed reservations have to be provided. Due to potential dependencies to other tasks these jobs have to be executed without any interruptions.

For enhancing the handling of problematic jobs in P, the partial execution has to be integrated into the SLA-scheduling algorithm. Instead of placing the job at the earliest possible place, where resources comply to the requirements of the SLA and where the entire job can be executed, the algorithm is also searching of appropriate gaps. This is the case if:

- the gap has at least as much resources as the job requests
- the gap is as least long enough for allowing the job to restart and checkpoint
- allocating the gap for partial execution of this job does not conflict SLA-adherence of any job in $\mathsf{S}^{\neg \mathsf{Res}} \setminus \mathsf{P}^{\mathsf{horizon}}$

If a gap complies to these requirements, the algorithm selects it for partial execution of the job. It then calculates the effective execution time of the job

within this gap (i. e. length of the gap minus time required for restart and checkpoint) and subtracts it from the remaining execution time of the job in question.

The algorithm then continues searching for further gaps, starting at the end time of the used gap. The search for gaps either ends if the entire job was successfully placed in existing gaps or the remainder of the job could be placed regularly at the end of the new schedule.

3.5.3.2 Increasing Utilization

The instrument of partial execution is not only useful for increasing the level of fault tolerance of a cluster system. It can also be used for increasing the overall utilization, applying it both to non-reservation SLA-bound jobs and best-effort jobs.

In both cases the SLA-scheduling algorithm has to be enhanced analogous to the described handling of problematic jobs. However, the check on conflicts with other SLA jobs can be omitted here, because no SLA jobs will be placed to the schedule at a later point in the algorithm.

3.5.4 Buffer Nodes

Despite on all efforts in using resources and even gaps within the schedule for compensating resource outages, the number of available spare resources is still the limiting factor for SLA-adherence. Depending on the number of failed resources and the general utilization of the cluster machine, the scheduler might not have sufficient resources for compensating the outage. In consequence, the resource owner is in danger of paying the penalty fees of all violated SLA contracts.

This situation is paradox, because a resource owner is primarily interested in a high utilization of his resources. But with increased utilization the system is no longer capable of compensating outages, resulting in penalty fees.

An approach in handling this situation is the introduction of buffer nodes to the system schedule. Up to now the scheduler treated all compute resources equally, placing jobs as soon as possible on an adequate compute node. By introducing buffer nodes the administrator is enabled to specify node allocation policies for a specified set of resources.

This way the administrator is able to block a number of nodes for executing SLA-bound jobs. This way, the scheduler would assign only best-effort jobs to these nodes. The SLA-scheduling algorithm does not have to be modified for this, because from the point of view of the scheduler only the number of nodes decreases while placing SLA-bound jobs. When placing best-effort jobs, the

scheduler may access the full range of resources, filling up the designated buffer nodes.

In the case of a resource outage, the scheduler would now be able to suspend or terminate all jobs running on these buffer node resources, which have runtime within the problem horizon. This is uncritical, because all jobs running on these nodes are guaranteed to be best-effort only. After freeing the buffer node resources, the scheduler may now allocate these resources within the regular scheduling algorithm.

Specifying buffer nodes significantly increases the probability that all SLA-bound jobs affected by the resource outage may be successfully completed on the buffer node resources. However, this method can only be successful if the number of buffer nodes is larger or equals the number of failed resources. Furthermore the buffer nodes may also be affected by a resource outage. In such a situation their buffer capacity is not available for handling problematic SLA-bound jobs. Therefore it is important to align the number of buffer nodes to experience values. However, it remains a trade-off between risk of penalty fees and the number of valuable SLA-bound jobs.

3.6 Fault Tolerance with cross-border scope

All mechanisms presented in section 3.5 enhance the level of fault tolerance of the resource management system. In case of resource outages, they try to use available resources in a way that guarantees the SLA-adherence of all agreed SLA. However, the number of available spare resources is always the limiting factor. Even with highly optimized algorithms, the number of available resources limits the options of migration.

Any attempt to significantly increase the level of fault tolerance therefore has to overcome this barrier. Since the resource management system can not increase the number of locally available hardware resources, it has to use hardware resources provided somewhere remote. This migration to resources outside the local cluster system is denoted as cross-border migration [38].

Once can distinguish between two different types of target resources:

- Resources within the local administrative domain: Often providers operate more than only one cluster system. By operating all of them with identical policies, they all belong to the same administrative domain, spanning over the cluster resources.

- Arbitrary remote resources: Grid systems allow users to access worldwide distributed resources in a transparent manner. Local resource management systems offer their resources to these Grid systems. By changing

the position of these RMS from passive job receivers to active Grid participants, local RMS systems can use arbitrary resources within the Grid, using the same access and transport mechanisms as the Grid end-user.

Both kinds of resources are qualified for resuming jobs from the latest checkpointed state, as long as specific requirements are met:

- Job-dependent policies: The migration of a job may affect non-disclosure issues. Therefore the owner of a job might insist of not migrating the cluster to any remote resource at any time, since he only trusts the chosen provider, but not arbitrary resource providers in the Grid. Hence, the customer may also specify only to migrate within the administrative domain, but not to external resources.

- System policies: Similar to the Grid customer, also the Grid provider has demands on the migration process, so that he might want to restrict the set of potential migration targets to a specific list of providers. The provider may also specify only to migrate within the internal administrative domain, but never to the Grid.

- Compatibility: A successful resume of a checkpointed job presumes the compatibility of the target resource chosen for migration with the source resource. In section 4.4 methods for ensuring compatibility are presented.

- Economic considerations: The introduction of SLAs should leverage the commercial uptake of Grid computing. Therefore customers as well as providers are following their particular commercial interest, focusing on their revenue in their business. The price for resource consumption and the penalty fee agreed within the SLA are determined in this context. Therefore a provider is only interested in migrating a job to a external target resource if this is economically reasonable.

Depending on these aspects the resource management system may start an external migration process. From the point of the scheduler there is no technical difference between migration to resources within the same administrative domain or resources somewhere within the Grid [5, 66]. The scheduler only decides which jobs are to be migrated to which resource and initiates the migration process. The technical details on migration (e. g. the negotiation with other Grid resources, following the data transfer using standard Grid file transmission protocols) are encapsulated in separate modules within the RMS.

3.6.1 Identification of Migration Candidates

The first step of migrating jobs to remote resources is the identification of potential migration candidates. Due to user or system policies, only specific jobs are eligible for being migrated to a remote system. This step does not yet evaluate the impact of a future migration of any steps, it only generates a preselection of potential jobs.

For this, first the provider's internal policy is evaluated. Here, the provider may specify:

- May jobs be migrated to resources in administrative domain respectively the Grid? If yes, in which order?
- Are there any exclude lists, defining which jobs may not be migrated? Are there any other constraints to be regarded (e. g. definition of maximum checkpoint dataset size for migration to Grid resources)
- Are there any include or exclude lists for target system selection (e. g. migration within the Grid only to specific providers)?
- May all jobs be migrated or only problematic jobs, or even only unhandled problematic jobs?
- Which jobs should be migrated first (e. g. high value, low value, most remaining time until deadline, most remaining runtime area, etc)?
- Should the system first try to migrate the affected problematic job, before trying to migrate other jobs?

The system analyzes all defined policy rules and matches them with SLA-bound jobs in the system. As result, the system gets a subset MT of jobs which might be migrated. For each of these jobs the system further knows restrictions or rules regarding the migration target.

This subset MT now has to be ordered. The primary sorting criterion has been defined by the system's migration policy, defining if problematic jobs from P should be migrated first, or if a migration is directly permitted also for currently running jobs, which are elements of set R.

From S only SLA-bound job describing reservations are of interest, since they have been already integrated at the time of handling problematic jobs in P. Non-reservation jobs are added to the schedule at a later point of time. Despite the situation given on internal fault tolerance mechanisms, reservation jobs may be migrated here, because it is possible to find a remote resource which is able to fulfill the reservation at the demanded point of time. However, these reservations are only added to the subset MT, if the reservation time span is

before or overlapping with the envisaged time frame of the problematic job (i. e. the demanded reservation time or deadline). If a reservation is located after all problematic jobs, its migration would not have any positive impact on the SLA-adherence of these problematic jobs.

The subset MT furthermore needs to be sorted by a second sorting criterion, which is also defined by the system's policy. By default, the value of the job is taken as second sorting criterion. The sorting order has to be increasing, so that unimportant or less profitable jobs are migrated first. This ensures that important jobs remain as long as possible on the source cluster.

If the administrator specified that only problematic jobs may be migrated which can not be handled, the set MT encompasses exactly these jobs. The second sorting criterion can then be omitted, since the scheduler will try to find suitable migration targets for all these jobs.

The system will not add by default SLA-jobs from R which are bound to fixed reservations, because this would interrupt the currently ongoing execution of the job and might cause side effects on other branches of a workflow (ref. section 3.5.2.2).

3.6.2 Application of Customer Defined Policies

Beside these system internal policies, the scheduler also has to take policies into account that have been specified by the customer, i. e. the owner of the job. These policies can be specified as part of the SLA request. Therefore these policies are subject in the SLA negotiation process and part of the contract between the service customer and the service provider.

The service provider has to ensure the adherence of all specified customer policies. Such a policy defines if the job may be migrated to a remote resource, and if it may be migrated to Grid resources or only within the same administrative domain.

The second step of the migration process consists of evaluating the customer defined policies for each of the jobs in ME. At this, prohibition always overrules admission. Therefore the customer defined policies overrule the provider defined: if the provider allows the migration, but the customer prohibits, the job may not be migrated. In this scope it is not possible that the system's policy is more strict than the customer's policy, because in this case the system's policy would have to forbid the migration. In this case, the job would not have been added to the set ME in the previous step.

This rigid handling may result in SLA-violations. In case the provider defined that only problematic jobs may be migrated that could not be handled by internal fault tolerance methods, the list of jobs for potential migration exactly equals the list of jobs which have to be migrated to match all SLAs. If the

customer defined policy of one of these jobs forbids all external migrations, the system has no option left for fulfilling this job's SLA.

The application of the customer defined policies may remove certain jobs from the set ME. Furthermore it may refine the restrictions or rules regarding the migration target, which have been set in the previous step.

3.6.3 Generation of Compatibility Profiles

At the end of the previous step the scheduler has an ordered list of jobs from R, S, and P which may be potentially migrated to remote resources according to the policy definitions of provider and customer. First in this list are problematic jobs, where policies demand to first try to migrate the problematic job itself. Following these jobs, the set contains all other jobs, which may be potentially migrated, sorted according to the provider policy.

Before the system is able to start any negotiation or migration process, it first has to create a catalogue for each of these jobs that describes the particular requirements on the target migration system. For this, it first takes all information that is known about the job from the negotiated SLA, e. g. number of nodes, memory size, or network interconnect. However, this is not yet sufficient to find compatible migration targets. Beside these basic demands the checkpoint dataset raises numerous additional demands (ref. section 4.4).

These demands are part of the compatibility profile of the job. The resource management system is able to generate such a profile by using appropriate information mechanisms of the process subsystem, as well as using RMS internal information retrieval mechanisms. The compatibility profile encompasses all relevant properties (e. g. name and version of required libraries) that are mandatory for successful restart of a checkpoint dataset.

3.6.4 Static Profile Matching

Having this compatibility profile catalogue information available for each job in the set, the system can now begin to look for potential target resources. For this, the scheduler will instruct a migration module within the resource management system to query for suitable candidate systems.

In the case of migrating within the same administrative domain, the system can follow the migration paths configured by the administrators. Since compatibility issues are known within a domain, the administrators may define that jobs from cluster A are always migrated to cluster B, because these systems are known to be compatible. Hence, the resource does not have to use the information from the compatibility profile, but solely follow the configurations.

The a priori knowledge about compatibility is not available when migrating to Grid resources. Here, the system has to query resource information services

within the Grid middleware, asking about systems complying with the demands of the compatibility profile.

For each of the jobs in the set, the system creates a list of potential migration targets. Depending on the system's and the customer's policies, these targets may be either located in the same administrative domain or somewhere in the Grid.

3.6.5 Filtering of Static Results

Even if all policies allow the migration of a job to a given compatible remote resource, it does not necessarily be reasonable to start the migration process. Before starting, the result set has to be filtered, taking the aspect of time for executing the migration process into account.

In contrast to intra-cluster migration, the checkpoint dataset has to be transferred to the remote cluster system. This transmission process usually uses public network interconnects, having significantly lower bandwidth as the inner-site network or even the intra-cluster network (ref. Section 3.4.3).

Before starting negotiations with target resources, the scheduler therefore first checks for each potential migration target, if it is possible to transmit the entire checkpoint dataset until the latest start time of the job. This point of time is determined by agreed reservation time frames or deadlines.

The scheduler may decide that a migration to Grid resources is not possible, because the available bandwidth on wide area network connections would not allow a timely data transfer. For these jobs, the scheduler would remove all Grid systems from the job's list of potential migration targets. The same holds valid for resources in the same administrative domain. Even having faster interconnects available within the domain, it may not be sufficient in case of close deadlines.

The filtering may remove all migration targets from the job's list. In this case the entire job is removed from the set ME, so that the system will not try to cross-border migrate it in the following.

3.6.6 Negotiation and Migration

All remaining migration targets remaining in the job's list are eligible for the migration process. First, the system handles all jobs, where the policy demanded to first try migrating the problematic job itself.

3.6.7 Migration of Problematic Jobs

For each of these jobs the scheduler instructs the migration manager module of the RMS to start an SLA-negotiation process with all potential target systems.

If at least one of these systems accepts the new job, the RMS now checks if the price of the cheapest SLA-offering is lower than the penalty fee that the provider would have to pay in case of SLA-violation. If none of the SLA-offers is cheaper, the resource owner may prefer to pay the penalty-fee.

If a reasonable priced offering has been received, the RMS accepts the SLA-offering of that system. After that it starts the migration of the job. Here the latest checkpoint dataset is transferred to the remote system and then restarted. After successful migration, the job is removed from the local system. In consequence, the job would also be removed from set P.

If no target system agrees on the SLA-request, it is not possible for the RMS to migrate it to a remote resource. Hence, the RMS can not directly ensure the SLA-adherence of this problematic job. However the system may still fulfill the SLA of the problematic job, if it succeeds on migrating another job to a remote resource, thus freeing additional spare resources for the execution of this problematic job.

3.6.8 Migration of General Jobs

After first handling problematic jobs, the system now continues on working on the set ME of potential migration candidate jobs. In contrast it does not directly starts on migrating job by job, but first evaluates the impact of each successful migration to the set of problematic jobs.

For this, the scheduler selects the first job of set ME and assumes that it can be migrated to a remote system. Working on a copy of the current schedule and based on this assumption, the migrated job is removed from the system and the scheduler generates a new schedule. The removal may have the following impact:

- No impact on problematic jobs: The releasing of the resources blocked by the migrated job in the schedule have not been sufficient to have impact on the problematic jobs.
- Set of problematic jobs has been decreased: The resources released by the migrated schedule resulted in removing at least one job from the list of problematic jobs.

If the set of problematic jobs has not been reduced, it is necessary to migrate additional jobs. Therefore the algorithm takes the next job from the set ME, assumes its migration and again evaluates the impact on the schedule.

Obviously it may be necessary that the algorithm iterates more than only once, before the set of problematic jobs is reduced. In such a case, it now enters the negotiation phase. Here it starts negotiation with all potential migration targets of all jobs which previously have been assumed to be migrated.

If none of the potential migration targets of a job is accepting the SLA-request, this job can not be used for migration. Therefore it is removed from the set ME and the scheduler has to recalculate the schedule, revoking the assumption that this job has been migrated. If the set of problematic jobs is still reduced, the algorithm can continue at this point. Otherwise it has to take the next job from ME, assume it has been migrated and again evaluate the impact of this migration on the schedule.

For all jobs having at least one migration target accepting the job, the scheduler takes the cheapest price of all SLA-offering of a job as the price for migrating this job. It then adds all these prices, having the total price for migrating these jobs. On the other hand it sums up the penalty fees that the provider has to pay if violating the SLAs of the jobs that were removed from P due to migration of the jobs.

If the penalty fee is lower than the price for migration, there is no point for the scheduler to start the migration. However it does not discard the already taken migration assumptions, but continues in taking the next job from ME, assuming its migration and again evaluating the impact on the schedule. To keep the assumptions is important, because by migrating on this basis could lead to the saving of a high-penalty job, which makes the entire migration process profitable again.

If the penalty fee is higher than the price for migration, it agrees on these SLAs with the job's migration targets. Following, the system transfers the job's checkpoint datasets and removes it from the system schedule. This then has the calculated impact, resulting in a smaller number of problematic jobs.

If set of problematic jobs is not empty after migration, the algorithm starts again by selecting the next job from ME, assuming its migration, and again evaluating the impact of this migration on the system schedule.

The algorithm ends if no problematic jobs remain or the remaining problematic jobs can not be solved by migration.

4 Design Aspects

In the report IST 2003 The Opportunities ahead, the European Commission reinforced that Grids will be important in achieving eEurope's goals for accessible services. They are central to the resource sharing that is an essential part of e-science, and also facilitate collaboration within virtual organizations. This should enable existing services to become more accessible and user-friendly and contribute to the creation of a new generation of services.

As part of the sixth framework programme (FP6) the European Commission published a call in 2004 on Grids for Complex Problem Solving, equipped with 52 million Euro of funding. The call focused on novel commercial and non-commercial opportunities arising from the wide field of Grid computing functionalities. An EC press release [26] states:

> By giving everyone access to the immense computing power and knowledge hitherto available only to the biggest corporations and laboratories, Grid tools will boost business competitiveness and help create new markets and services.

This mission statement perfectly matched with the core statement of this work: if the commercial user should be attracted for the Grid in a way that new business models can evolve, local resource management systems have to provide contractually fixed levels of service quality. After partners were found for key working fields, a project proposal has been created based on these ideas, and submitted to the EC.

This project was named HPC4U (Highly Predictable Clusters for Internet Grids) and accepted for funding by the EC in 2005. In this project the Paderborn Center for Parallel Computing is working on providing SLA-aware Resource Management. In addition the technical partners IBM, Seanodes (both France), Scali, and Dolphin (both Norway) provide their services on process checkpointing, storage, and network to the RMS. Non-technical partners CETIC (Belgium), Fujitsu (France), and University of Linköping (Sweden) complement the project with work on verification, validation, dissemination, and exploitation.

Large parts of this work have already been implemented and published in the scope of this project. This chapter will highlight the concepts of HPC4U and present its architecture, core components and main functionalities.

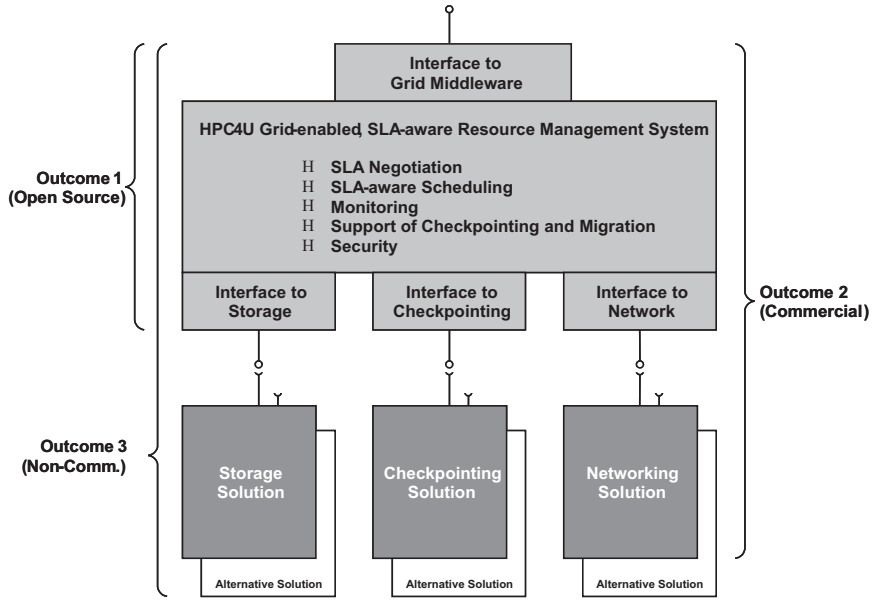

Figure 4.1: Outcomes of HPC4U

4.1 The HPC4U Project

The goal of HPC4U is to provide a software-only solution for a transparent and reliable cluster middleware. HPC4U allows the Grid middleware to negotiate on SLAs, which is realized at the cluster middleware level by means of HPC4U's main fault tolerance building blocks: process checkpoint, storage snapshot and virtualization, and network failover.

The HPC4U cluster middleware consists of multiple elements, i. e. the SLA-aware resource management system and the main building blocks for ensuring a high level of fault tolerance: process checkpointing, storage snapshot and virtualization, and network failover. In an exceptional situation, e. g. the outage of hardware resources, the HPC4U system uses its FT (Fault Tolerance) mechanisms to assure the completion of a job. This means that the Metacluster software enables checkpoint/restart (and migration) of a running process, so that jobs can be restarted from the last checkpoint on a spare resource. But only considering the checkpoint process could cause inconsistencies at restart, because the checkpoint's data and job's data can be at a different stage as a running job continues to write data on files after the checkpoint. Therefore, the system has to maintain consistency between checkpoints' data and job's data. This process has also to be supported by the network subsystem, e. g. regarding in-transit network packets.

The results of HPC4U are a mix of open source and proprietary software embedded in two outcomes (cf. Figure 4.1) [27]. The SLA-aware and Grid-enabled

Resource Management System includes SLA negotiation, multi-site SLA-aware scheduling, security and interfaces for storage, checkpointing, and networking support. It is available for multiple platforms and distributed as open source. The second HPC4U outcome is a vertically integrated commercial product with proprietary Linux-specific developments for storage, networking and checkpointing. This outcome demonstrates the entire, ready-to-use HPC4U functionality (job checkpointing, migration, and restart) for Grids based on Linux architectures. It is obvious that providing an agreed level of Quality of Service and Fault Tolerance requires broad interaction between all components of the HPC4U system. The third outcome also depicts a vertically integrated system, but consisting of non-commercial components only. Compared to the commercial system this system has significant functionality drawbacks, but can be easily evaluated without the need of obtaining any licenses.

Without loss of generality we assume that a user from somewhere in the Grid wants to compute a job and connects to an HPC4U system for negotiating on a Service Level Agreement. Usually, a user would not connect directly to an HPC4U system, but uses his local Grid middleware interface for finding suitable resources for his request. Matchmaking mechanisms on the level of Grid middleware compare requirements with published information about available resources. Hence, Grid middleware mechanisms offer intermediary services. However, from the point of view of an HPC4U system, it makes no difference if a user or some Grid middleware element starts a service negotiation request (cf. section 2.3).

The cluster middleware system consists of three independent layers:

- At the upper layer, the system provides an interface, which can be used by Grid middleware systems to negotiate on Service Level Agreements. This interface applies to standard protocols used in Grid middleware ensuring interoperability with other projects.

- At the middle layer, an SLA-aware resource management system using the upper layer interface, negotiating with customers on SLAs. It also assures the compliance with these agreed SLAs at runtime. This does not only imply the monitoring of internal resources, but also the utilization of appropriate mechanisms to realize fault tolerance in case of resource outages.

- At the lower layer there are the subsystems of HPC4U. Offering specific APIs, each of these subsystems provides special mechanisms for fault tolerance on process-, network- or storage-level. Since all interfaces within the HPC4U system are standardized and published, each component can be replaced with arbitrary third-party products, as long as these products provide compliant interfaces.

In the following, the resource management system will be explained in more detail.

4.2 Resource Management System

As it has been stated above, the resource management system (RMS) plays a central role within the HPC4U architecture [37]. Since it is an SLA-aware RMS, it has to keep an overview about all SLA-related activities within the HPC4U system. First of all, it has to negotiate with customers on new service level agreements. These users may be located somewhere in the attached Grid system, accessing the SLA-negotiation interface of the HCP4U cluster middleware system. The RMS first decides on starting a negotiation process (a negotiation request may be rejected due to local policies), then actively negotiating on the contents of the requested agreement. In this negotiation process, the current system condition has to be considered.

Therefore it is necessary that the RMS has the complete overview about the HPC4U system. This encompasses available resources (e.g. number, type and equipment of compute nodes, topology and characteristics of the interconnect between these compute nodes, or characteristics and capabilities of the available storage system). Beside these static aspects, the RMS also has to be aware of dynamic attributes. The validity period of such dynamic attributes normally is really short, as these aspects represent the current condition of the overall system.

The resource management system is also responsible for planning not only the current system usage, but also the future. Therefore it holds a schedule of all accepted SLA-bounded jobs. According to this schedule, the general static information, the dynamic information representing the current system condition, and the requirements of the new SLA request, such a request will be accepted or rejected.

It is important to stress that the resource management system is the only element of HPC4U which has direct contact with the Grid system. The subsystems of HPC4U are only contacted by the RMS, but not from the Grid user.

To be able to plan requests with assigned SLAs an RMS scheduler not only has to count free resources, it also has to respect system specific constraints like the topology of a high speed network. The RMS used in the HPC4U project is CCS (Computing Center Software) [11, 63, 7]. It does this by splitting the scheduling process into two parts, a hardware-dependent called Machine Manager (MM) and a hardware-independent part called Planning Manager (PM). This separation allows to consider system specific requirements (e. g. location of I/O-nodes or network topologies) at which the MM part may be adapted to different resource configurations without changing the basic scheduling part

(i. e. the PM). The MM verifies, whether or not a schedule computed by 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. node availability) information on the system resources. Information provided by the subsystems is incorporated in this mapping procedure. If the MM is not able to find an SLA conform mapping of the jobs onto the resources at the time scheduled by the PM it tries to find an alternative time. The resulting conflict list is sent back to the PM which in turn accepts the schedule or computes a new one based on the schedule given by the MM.

SLA-aware RMSs are responsible for fulfilling the statements of all SLAs, which have been agreed. This implies that the RMS has to take appropriate actions in case of resource outages [8]. Hence, jobs have to be observed during their whole lifetime. For this purpose, the MM monitors the running jobs and the affected resources. In case of an error the MM is able to migrate the job to another matching resource. Since the MM always knows the current schedule this may be done without violating the current schedule. In HPC4U, the RMS utilizes the functionalities of the underlying HPC4U subsystems to provide fault tolerance, thus guaranteeing the adherence with the negotiated SLAs.

The RMS also has interfaces to the three mentioned HPC4U subsystems. These interfaces are located in the MM part of the RMS since the MM also controls the execution of jobs. Using the interfaces the RMS may subscribe callback routines which are called in case a subsystem notices an error and is not able to solve the problem alone and therefore needs help by the RMS. For instance we assume a job with an SLA which guarantees a minimal bandwidth. It may now happen that due to a resource failure (not necessarily used by the concerned application) the networking subsystem is impelled to change its routing tables. The subsystem tries to fulfill the mentioned SLA since the RMS started the application with this constraint. If the networking subsystem is not able to keep the agreed minimal bandwidth it informs the RMS about the problem. The RMS then has to decide what to do: migrating the concerned application or suspending another one.

The resource management system consists of numerous sub-components (ref. Figure 4.2). The service of all these subcomponents has to be orchestrated, so that the adherence with agreed SLAs can be ensured. Within this architecture, the Planning Manager is of central importance. In the following, the tasks and interaction between these blocks is to be explained.

4.2.1 Grid Interface (GI) with Negotiation Manager (NM)

The HPC4U system is able to act as an active Grid component. This implies that the system not only can be used to process incoming jobs from the Grid. The system can also use the Grid for improving its level of fault tolerance. If

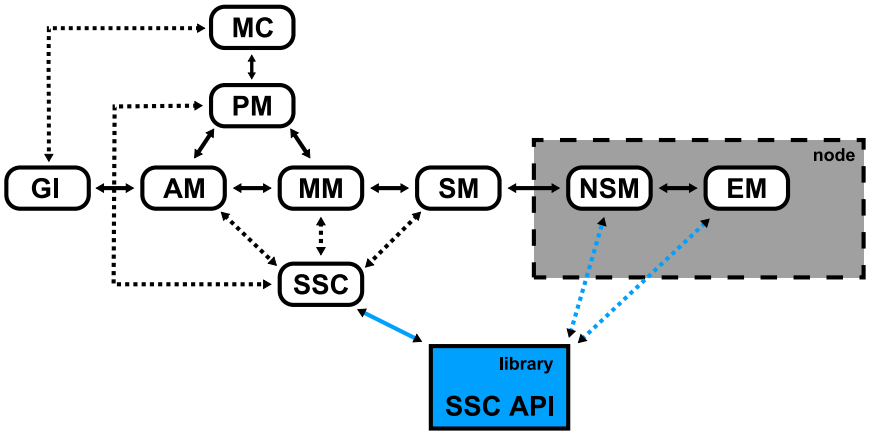

Figure 4.2: Overview RMS Modules

no internal spare resources are available in case of resource outages that would allow the adherence with the given SLA, the RMS is able to migrate the affected job to Grid resources.

This interaction with the Grid is encapsulated within the Grid interface (GI), having the Negotiation Manager (NM) as subcomponent. This interface can be aligned to the used Grid middleware system, so that the RMS can be integrated into Grid infrastructures without the need of changing the configuration of other modules. The Negotiation Manager steers the negotiation process with incoming user requests and communicates with the Planning Manager. Only if the Planning Manager is able to fulfil the requirements of the SLA, the Negotiation Manager accepts the request.

The same holds valid for outbound requests. All communication interaction is encapsulated within the Grid interface. According to the attached Grid Middleware system, the Grid interface has to provide appropriate mechanisms and to support required protocols. In case of resource outages the FT Manager and the Planning Manager uses the Grid interface for querying the Grid for appropriate spare resources. If such resources can be found (i.e. the Grid provider assures the contents of the SLA) the job is transparently migrated.

4.2.2 Planning Manager (PM) and Machine Manager (MM)

In the design of the resource management system, a compromise has to be found between two conflicting goals: on the one hand the design of the RMS should utilize available resources (e.g. processors, network, and storage) optimally. On the other hand, the system should not be tailored to one specific configuration or technology, but be able to be deployed and ported to arbitrary usage scenarios.

The scheduling process is the central component of a planning based resource

management system. This component decides which jobs are accepted for computation and which guarantees (e.g. resource reservations) are given. For realizing a portable system which utilizes systems optimally, the scheduling process has been split up into two distinct and independent modules.

The Planning Manager (PM) is the hardware-independent part. It has no information of any mapping constraints, such as network topology or exact location of I/O nodes. The PM checks if the jobs can be planned according to their general statements, not regarding static or dynamic information of the hardware machine. The result is a schedule which is coherent with the requirements of the received and processed requests. This proposal is then handed over to the Machine Manager (MM), which does the hardware-dependent mapping of the schedule to the target machine. The MM provides machine specific features, e.g. partitioning of systems into subsystems, or job controlling. Thus, the MM is hardware-dependent. It verifies whether a proposal for a schedule of the PM can be mapped onto the available resources (e.g. compute nodes, network, and storage) at a given time. The MM checks this by mapping the user specifications and requirements with the static (e.g. static topology data) and dynamic (e.g. current resource outages, current load situation) information of the system resources of the cluster machine.

The MM allows to implement mapping modules that are tailored to the specific hardware configurations and properties of the target cluster machine. For instance, specific characteristics of the network interconnect can be supported easily by adding additional functionality to this module. Furthermore, this makes it simple to adopt the system to future technologies in high performance computing.

This separation of planning to PM and MM includes the consideration of dynamic and static information. If the MM is not able to map the proposed schedule to available resources (e.g. due to a resource failure), it sends back a counter proposal of the schedule to the PM, which checks this counter proposal in the light of the requirements of received jobs. This process iterates until a valid solution has been found.

If no schedule could be found, this either implies that an incoming request has to be denied, or that an accepted job has to be cancelled. At this point, HPC4U enriches the RMS mechanisms. Instead of cancelling a job (e.g. a job has to be cancelled due to a resource failure, since the MM cannot provide suitable resources anymore), the RMS can use the fault tolerance mechanisms of the HPC4U subsystems to ensure the adherence with the accepted agreement.

It is a good strategy to solve problems as local as possible. Particularly regarding a job with huge checkpoint and snapshot datasets, it is expensive or impossible to migrate it within a reasonable time frame to a remote resource. Therefore the PM first tries so solve exceptional events (e. g. the outage of a compute nodes) by means of the mechanisms of the HPC4U subsystems,

executing specific commands via the HCP4U subsystem controller (SSC).

4.2.3 Access Manager (AM)

In most cases, the access to a compute resource is restricted. Especially in cases where valuable resources are provided, providers charge their users according to their resource consumption. Administrators of such machines may also follow the policy of prioritizing local users, so that only unused idle compute time is made available to other (i.e. foreign, non-local) users.

The notion of accounting describes the procedure of granting access to a specific compute resource for a requesting user. In the case of the RMS of HPC4U, the Access Manager (AM) is in charge of this task. First it checks the authentication of users requesting for compute resources. Since the AM may be modified easily, arbitrary authorizing mechanisms can be supported and implemented. In case of a project based authorization, the user is assigned to specific projects, each of them having specific access rights on the machine. In such a scenario, the user authenticates using his password and project name, following an authorization check of the RMS. Only if the password of the user is valid and registered for the specified user, his request is be further processed.

In this case, the AM checks accounting information, so that the user or the specified project can be charged for resource consumption. The RMS also allows the specification of budgets, so that the usage of a machine for a given project can be limited. If a user tries to request compute time exceeding the given budget, the request is rejected.

The AM is also responsible for ensuring and enforcing access policies. Such policies define which users and user groups are eligible to access which parts of the system at what time. The AM checks if the request is coherent with specified policies.

4.2.4 Migration Controller (MC)

As stated in the section 4.2.2, PM and MM always try to solve resource outages internally. This implies that these modules decide on using internal fault tolerance mechanisms, e. g. the generation of checkpoints. If an internal handling of problems is not possible, since the PM is unable to reschedule in a way that the SLA-adherence of all agreed SLAs can be ensured, the PM can try to migrate jobs to remote resource. The entire process of migrating jobs to remote resources is executed by the Migration Controller (MC).

Regarding constraints following from provider defined policies as well as customer defined policies, the migration controller also tries to solve problems as local as possible. Since clusters within an administrative domain normally are connected with high speed networks, the MC first tries to solve the problem

"locally". Therefore it queries other systems within the same administrative domain for compatible resources for a job. If such resources can be found, the checkpoint and snapshot datasets can be transferred to the compatible local resource and be restarted there.

If the querying for resources within the local administrative domain failed, the MC has to locate suitable resources on other sites. The Grid Interface (GI) module of the RMS is the interface of the entire HCP4U cluster middleware to the Grid. The MC now utilizes the functionalities on resource information retrieval to query for suitable spare resources. If this query process returned potential candidates, a negotiation process on a new SLA is initiated, eventually followed by a transmission of the checkpoint dataset. This entire process is driven by the MC, but mapped to Grid protocols and services by the GI.

4.2.5 Configuration Manager (CM)

In SLA-aware resource management systems it can be distinguished between different phases of operation (cf. section 3.3). In the pre-runtime phase the set of assigned compute resources of a job has to be configured. This is a pre-requisite for executing the job according to all the demands of the agreed SLA. Accordingly, the resource has also to be reconfigured after job completion. These configuration tasks may comprise the configuration of the storage subsystem or the initialization of the network interconnect. In case of HPC4U, basically the subsystems have to be configured and initialized, so that the fault tolerance mechanisms can work.

The configuration manager is responsible for these tasks. It is invoked before the job is started on the resources, respectively if the computation has been concluded. In case of resource failures, it may also be started at failure handling, as spare resources have to be configured.

In cluster machines, access control is of central importance, because each node runs a full fledged operating system. Therefore it could be used as a stand alone computer by its owner. When operated in cluster mode, users must be prohibited to start processes on single nodes, not only because of unpredictable changes in CPU load, but also because they might create orphan processes which are quite difficult to clean up by the resource management system.

Hence, the resource management system also has to care about the access- and job-control. This is done by dynamically modifying system configuration files. The exact modifications are depending on the installed operating system. By this, the exclusive access can be granted to the temporary node owner, i.e. the owner of the job which is to be computed on the compute resource. Before releasing the partition, pending processes and files have to be removed. Also the configuration of the node has to be modified, so that no user is able to login into this node. Otherwise it would be possible to misuse the node, or at least

bypassing the control of the RMS. Once this node is assigned to a new job, the configuration is altered, allowing the owner of the new job to login into this node.

4.2.6 Execution Manager (EM)

The execution manager (EM) has to start the job, which has been submitted by the requesting user. By this, the EM represents the job execution level. It sets up an execution session on the target compute resource, performs application-dependent pre- and post-processing, establishes the user environment (e.g. shell environment, shell settings, and environment variables) and maintains the status of the application.

In the case of HPC4U, the EM also utilizes basic FT mechanisms of the underlying HPC4U subsystems. By this, it utilizes the checkpointing subsystem for starting the process in a virtual bubble, so that it can be checkpointed at a later time. The EM is also responsible for executing local commands, which are necessary for FT provision. For example, EM executes the command for creating checkpoints, respectively for resuming the job from a checkpointed state.

4.2.7 Subsystem Controller (SSC)

Flexibility has been a major design goal of the integration of the HPC4U subsystems into the resource management system. Even if the main outcomes of HPC4U mainly focus on the products of the HPC4U partners, the interface should be general enough to also allow the integration of other products. This flexibility is also required within the HPC4U project, since the integration of CCS with non-commercial third-party products for checkpointing is required for the freeware outcome.

The architecture for integration of the HPC4U subsystems consists of the following core elements:

- The Subsystem controller (SSC) is a regular CCS daemon, which receives the monitoring information of the subsystems. It may also actively poll other CCS daemons for retrieving necessary information. It initiates commands by sending messages to other related daemons (e. g. the Planning Manager). Moreover, it initiates central commands (e. g. establishing of a checkpointing bubble).

- The SSC-API is an interface which is mapping the internal commands of CCS to the respective commands of the used products in the subsystems. The API is used by a number of CCS daemons, e. g. the Node Session Manager (NSM), Execution Manager (EM) or the Access Manager (AM).

The SSC itself provides high level functions to the upper layer modules, e. g. the generation of a new checkpoint. Within the SSC this high level functions are realized as workflows, containing single tasks which have to be mapped to the particular environment. As an example, before generating a new checkpoint of a running process, the system first has to check for valid licenses and available checkpoint directories.

The SSC is configured over a central configuration file, holding all information for all subsystems of HPC4U. Within this file three main parts can be distinguished:

- Order of execution
- Mapping of commands
- Mapping of return codes

For some high level commands it is necessary to execute commands in more than only one subsystem (e. g. for checkpointing a running job, the process as well as the storage and the network has to be checkpointed). Depending on the used software solutions within the subsystems, the order of execution in these subsystem differs. In case of the commercial HPC4U stack it is sufficient to first checkpoint the storage, followed by a checkpoint of the storage. It is not necessary to checkpoint the network, since this operation is automatically initiated by the process checkpoint operation. In contrast, in the non-commercial outcome the RMS first has to execute the checkpoint command within the network subsystem.

The configuration file allows to define the order of execution for each task. At runtime the SSC analyzes the configured order and executes the particular subsystem dependent low-level commands accordingly.

The second main part of the configuration file relates to this very execution of low-level commands. Depending on the used software system within the subsystem, product specific commands have to be executed, each expecting different kinds of input parameters. While the commercial process checkpointing solution expects the abstract job ID for identifying the job on checkpointing, the non-commercial process checkpointing system expects the process ID.

Due to this reason the SSC provides a wide range of different parameters that can be added in form of placeholders into the command mapping here (e. g. "%JOBID" for the ID of the job, respectively "%PID" for the process ID). At runtime SSC replaces these placeholders during the command mapping operation with the current values. This provides the flexibility to integrate arbitrary solutions within the subsystems.

The third part of the configuration file is important for analyzing the success of the executed functions. Each command in every software system returns

different error codes. Most systems are only consistent in the fact that zero as error code means "success", while anything else means "error". For the SSC it is important to be able to distinguish between the error codes, so that it can react accordingly (e. g. if the subsystem returns error code "17", SSC should recognize that this signals a bad license).

The configuration file allows a subsystem dependent mapping of codes to common CCS error codes. This way the administrator is enabled to define that error code 17 in the process subsystem is mapped to `CCS_ERROR_NOLICENSE`.

By shipping the resource management system with a default set of configuration files for standard checkpointing, storage, and network systems it is easy for the administrator to deploy the system in his environment. Depending on the used system, the administrator then only has to merge the fragments contained in the deployed CCS version, creating his own SSC configuration file matching his installed system.

4.3 Checkpoint Generation

The generation of checkpoints from running application is the core functionality for providing fault tolerance to the running application. Without any checkpoints available, the results of a running application are lost in case of resource outages. In the light of long running jobs, potentially running over weeks, using numerous computers of a cluster in parallel, the total number of lost compute hours is immense. Therefore the RMS regularly checkpoints the running application (e.g. one checkpoint each 60 minutes). In case of a resource outage, the application can be restarted from the latest checkpointed state, such that a maximum of 60 minutes is lost.

Since the checkpoint dataset must encompass not only the process memory and state, but also the network and storage, an orchestrated operation of all subsystems is mandatory to ensure consistency. This operation is depicted in figure 4.3.

The first step in the generation of a new checkpoint is an interaction between the RMS and the process subsystem. Here, the RMS requests the process subsystem first to suspend the process, such that it does no longer change it's state nor process any operations. At this state, the process subsystem may start to generate the process checkpoint dataset, either by using the IBM Metacluster system or the Berkeley Lab's BLCR system.

In the case of MPI-parallel applications it is not sufficient to solely checkpoint the state of the application on a single node. Here, it is essential to checkpoint all processes of the application running on all used nodes (these nodes are denoted as partition of the parallel job). Since MPI-parallel applications are communicating via messages sent over the network infrastructure, it is

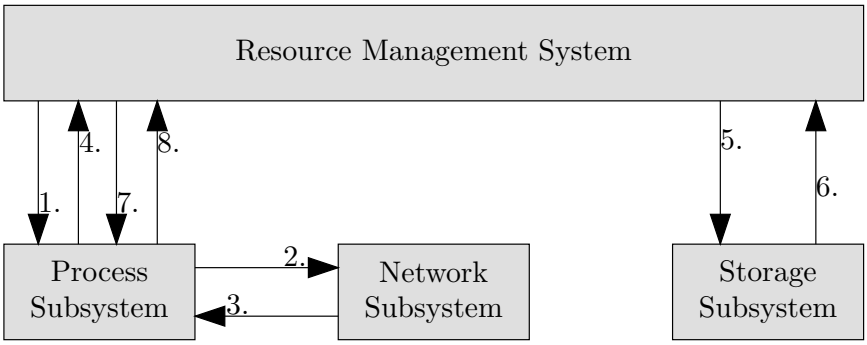

Figure 4.3: Generation of a Checkpoint

also essential to checkpoint the state of the network at checkpoint time. Among the checkpointed data are the network queues on all nodes of the partition as well as in-transit-packets (i.e. networks currently “on the wire"). This network checkpoint process is performed in the second step of the checkpointing process.

Since the generation of the network checkpoint is time critical and has to be performed in a narrow time interval after the process checkpoint to avoid timeouts, the Cooperative Checkpoint Protocol (CCP) has been introduced, a direct interface between the process subsystem and the network subsystem. This way, IBM Metacluster (used in the process subsystem) and Scali MPI Connect (SMC, used in the network subsystem) are able to orchestrate their operations at checkpoint time.

In the case of the LAM-MPI as non-commercial alternative to the commercial HPC4U stack, the CCP protocol is not used. LAM-MPI has daemons running on all nodes of the partition, called LAMD. The daemon processes are responsible for starting and terminating the processes of the parallel application on the specific node. LAM-MPI provides System Service Interfaces (SSI), which can be used to provide additional services to the MPI implementation. Among others, one SSI implementation is responsible for realizing the checkpoint and restart of MPI-parallel applications. The mpirun process, responsible to start up the MPI-parallel application on all nodes of a partition, initializes the SSI modules responsible for BLCR checkpointing on program startup. Once a checkpoint should be generated, these callbacks are used. First, the mpirun process has to write the topology of the MPI-parallel job. After this, signals have to be send to the nodes of the partition, such that the MPI-parallel processes of the job are also saved. For this, it uses the LAMD daemon processes, which initiate the checkpointing of the process on the particular node.

It becomes obvious, that the HPC4U commercial solution consisting of IBM Metacluster and Scali SMC follows a different approach as LAM-MPI and BLCR. In Metacluster/SMC, the process checkpointing solution is the driving force in the checkpointing process. It steers the process and signals the

network subsystem to initiate the checkpoint of the network state. In contrast BLCR/LAM-MPI is driven by the network part, which is responsible for the orchestration of the entire process. The BLCR is solely the passive part, responsible purely for checkpointing processes. However, for the RMS, this difference is transparent, since it is only signaling to the process subsystem, that the checkpoint is to be generated, not caring about the following execution. The process subsystem is configured to the respectively used system (i.e. either Metacluster/SMC or BLCR/LAM-MPI) and executes the required commands for checkpointing.

In the third step, the process subsystem signals back the result of the preceding checkpointing efforts regarding process and network. In case of any errors, the RMS now has to decide to repeat the checkpoint (e.g. if the checkpoint failed due to insufficient disk capacity) or to skip this checkpoint (e.g. if the checkpoint failed due to temporary problems). If the checkpoint failed due to permanent problems (e.g. missing or expired licenses), the RMS may decide to terminate the job (in case checkpointing is mandatory for the process) or to resume without checkpointing. Even if the execution of the application may resume, failures in execution of checkpoints (both temporary and permanent) are critical for the RMS, since fault tolerance mechanisms base on the existence of checkpoint datasets.

Presuming the successful execution of the previous step, the RMS now has to ensure the checkpoint of the application's storage. This is crucial, since a consistent image is required at resume time, consisting of process and network as well as storage. This is the task of the storage subsystem, which is signaled by the RMS in the fourth step. This step is not time critical, since the application is still halted, such that it does not change its output files.

In the case of the commercial outcome of HPC4U, the storage solution Exanodes of Seanodes is used. Exanodes realizes the virtualization of storage capacity of cluster disk drives. In a cluster system, each node usually has more disk capacity than required, such that a major part remains unused. Exanodes uses these unused disk partitions and establishes a virtual volume. The user or administrator now can request storage capacity from this virtual volume, without knowing which parts of which real partitions of which nodes were actually used. Using the RAIN (redundant array of independent nodes) [6] technology (which is similar to the widely known RAID - redundant array of independent disks), Exanodes is even capable of dealing with node outages. By introducing a certain level of redundancy, Exanodes is able to recover storage capacity affected by a node outage. Exanodes is also able to generate storage snapshots (i.e. checkpoint of storage). On requesting storage, the user is able to specify certain parameters (e.g. size or QoS characteristics). Exanodes then provides a storage container, which is a partition on the virtual disk. The user now can format and mount the container like a typical block device. If executing a snap-

shot, the Exanodes system saves the state of the storage container, such that the content can be reproduced at a later point of time.

For the non-commercial outcome of HPC4U, the storage subsystem uses traditional storage. However, it is possible to realize fault tolerance, e.g. by using RAID5 disk arrays. Since these systems do not provide storage snapshot functionality, the storage subsystem has to save the content of the storage part of the running application using programs like rsync or by simply creating a tar-ball.

In the sixth step, the storage subsystem signals the result of the preceding storage snapshoting back to the RMS. Similar to the fourth step, the RMS now has to react on failures. Again, it can decide to resume or abort. It has to be highlighted that fault tolerance can only be realized if both process/network and storage have been successfully checkpointed. If one of them has failed, no consistency at restart time can be ensured.

After checkpointing process, network, and storage, the RMS signals to the process subsystem in the seventh step that the execution of the application can be resumed. The successful resume is then finally signaled back to the RMS, which can update its internal databases concerning the existence of a valid checkpoint of the particular application. This dataset can be used at a later point of time to resume the application in case of failures, e.g. resource outages.

4.4 Compatibility Profile

The SLA-aware resource management system uses process checkpoints for various purposes. Beside the compensation of local resource outages by intra-cluster migration (cf. section 3.5), a checkpoint may also be transferred to remote systems (cf. section 3.6). Since the provision of fault tolerance must be transparent for the running application, application- and user-level checkpointing solutions are inappropriate in this context (cf. section 6.2).

Kernel-level checkpointing systems allow the checkpointing of arbitrary application without the need of prior relinking or recompiling. Focusing commercial users executing their commercial code, relinking or recompiling would not be possible in most cases anyway. This flexibility and transparency on the other hand has the drawback of a high grade of system dependence.

In contrast to application level checkpointing, a checkpointed process can not be restarted on arbitrary target systems. Beside high level characteristics like operating system or processor type, the target machine even has to be compatible in regard of versions of installed libraries and tools. If restarting a checkpointed job on an incompatible resource, the job would directly crash at best. In the worst case, the application would resume its computation, but return incorrect results. In this case the RMS would assume that the application restarted successfully, returning incorrect results back to the customer.

An obvious way to face this situation and ensure a successful restart on the target machine is to request identical machines. At this, the hardware of a suitable target machine must be identical to the source machine. The same holds valid for the software installation. Both machines have to have identical operating systems with identical upgrade levels (e. g. RedHat AS4, Upgrade 4).

Even if this strict demand solves the problem of compatibility very efficiently, it reduces the number of eligible target systems in a migration process close to zero. If looking to resource information catalogues in the Grid, a broad variety of different systems becomes apparent. Even if some of these systems would be able to resume the checkpointed application, this strict demand on equality would disqualify them.

For enhancing the number of potential migration targets while ensuring their compatibility, the compatibility profile is introduced. This profile is an instrument for describing the application's requirements on the target machine, so that the restart can be successful. In the following, the different parts of this profile will be described.

4.4.1 Architecture and System Properties

The most fundamental requirement on the target machine is regarding its internal architecture and system properties. These demands are not specific to the used checkpointing system, but arise from the execution environment.

4.4.1.1 Operating System

The operating system installed on a compute node forms the fundament for the application execution. Most systems in current Grid environments are running the Linux operating system. However, also other operating systems like OpenBSD [82], Solaris [94], or Darwin (Apple MacOS X) [18] exist. Even Microsoft is pushing for increasing the role of Microsoft Windows operating system in the HPC domain. The recently released Microsoft Windows CCS [69] nowadays still is an exotic platform for HPC or Grid computing, but Microsoft undoubtedly has the means to also enter and dominate this market.

In this work only the Linux operating system has been targeted. But also here it is essential to distinguish between the different versions of this operating system. Even if there are only minor differences in the kernel version between two systems (e. g. migration from Linux kernel 2.6.9-11 to 2.6.9-34), these kernel versions may have important differences in core functions of the system.

During the migration process, this attribute is easy to evaluate, because most providers publish the exact kernel version of their compute nodes to resource information services in the Grid. Since a kernel version in many cases is closely

related to core system libraries, this attribute also represents a preselection criterion for these libraries.

4.4.1.2 Checkpoint System

Kernel-level checkpointing solutions all have their general functionality in common. By intercepting specific system calls they allow to generate a process image of a running application. Despite the fact that these solutions differ in their particular functionality profiles, it is not possible to exchange checkpoint datasets between them. Therefore it is necessary to have the same checkpointing solution available on the target machine that was used to generate the checkpoint.

Moreover, the version of these checkpoint system is also essential, because new versions often introduce new features. This has implications on the internal data structure of the written checkpoint dataset file, differing from version to version. Unfortunately most checkpoint systems neither write versioning information to their checkpoint dataset files, nor do they support backward compatibility on restart. Therefore it is not only essential to have the same checkpoint system at restart, but also the same checkpoint system version.

These values are easy to configure within the configuration of the resource management system. Unfortunately information about the checkpoint system is not yet propagated in resource information systems within the Grid, so this aspect has to be added as an item within the SLA-negotiation process.

4.4.1.3 Processor Architecture

The architecture of the processor has to be identical in source and target machine. It is technically impossible to start the image of a Pentium-based process on a processor like IBM's PowerPC. Even if some processors support the execution of legacy code (e. g. AMD's 64bit processors allow the execution of Intel 32bit code), it is in general impossible.

For describing the architecture the standard tag could be used, as reported by the Linux operating system (e. g. x86 for processors compatible to Intel's 32bit processors). If the operating system only gives information about the actual processor (e. g. AMD Athlon), this has to be mapped according static configuration files.

Just like the information about the operating system, this property is easy to match during the migration process. Most providers report the processor architecture or the actual processor installed in their machines. Therefore this criterion can be evaluated without direct interaction with the target provider.

4.4.1.4 Other System Properties

Beside operating system, checkpointing system and processor architecture, a broad variety of other system properties is essential for a successful restart of the application.

First of all, the target machine has to have sufficient amount of main memory as well as storage capacity. The demand of the application can not be determined from the actual resource usage, since this may change at a later point of application execution. However, these parameters are part of the SLA negotiated with the job owner. In this SLA the provider ensured to provide a machine that complies with these requirements (e. g. 4GB main memory). If the application crashes due to a memory allocation error at runtime, even if the execution host has the required amount of memory, the problem is caused by underestimation of the customer.

The same holds valid at migration time. If the source system ensures that the target system complies to these requirements of the agreed SLA, the target resource is appropriate for restart.

Also hardware demands like the availability of a specific network interconnect, or software demands like special purpose applications, libraries, or licenses are part of the customer agreed SLA. The SLA may also demand for the availability of a specific filesystem type. In this case, also the filesystem must be available on the target machine.

These properties usually are not listed in resource information systems in the Grid. Therefore the resource management system has to add these properties to the SLA request sent to the potential migration target. If the target system agrees on that SLA, the RMS can be sure that the application resumes as desired.

4.4.2 Libraries

The concept of libraries is known in almost all operating systems. Instead of demanding each application developer to write the same core functions again and again, these functions are provided by means of libraries. The operating system itself is offering system services over system libraries. By linking his application against these libraries, this the programmer is able to use these functionalities easily.

Static libraries are linked to the application and part of the resulting binary. This way, the user does not have to ensure the availability on the system where he plans to execute the binary. However, static linking results in significant waste of space, both storage and memory. Furthermore this type of library complicates system maintenance. On updating a given library (e. g. due to a security problem or programming bug), all applications using this library have

to be relinked.

Shared libraries in contrast are only loaded once into the system memory. On application start, the availability of the library is checked by a loader service. This loader verifies the version of the library, sets entry addresses and maps the memory of the library to the virtual memory segment of the application.

Dynamic loading further improves the concept of a shared library. It allows the application to dynamically load and unload a library at runtime. Beside performance increase at start time, this method also has the advantage that applications can start even if specific libraries are not available on that system.

From the checkpoint compatibility point of view dynamic loading is a serious issue, because libraries are not necessarily placed at the same position in memory at each restart. If a checkpoint is resumed in an environment where these libraries are loaded to different memory addresses, the application would access the wrong memory segments at runtime.

Metacluster solves this problem by saving the address of these libraries to the checkpoint dataset file. If restarting the application on a remote system, Metacluster checks the addresses of these libraries. If necessary, it then reloads the library and maps the addresses for the restarted application. However, this method of Metacluster requires the library installed at the same position (i. e. directory path) and in the same version. A similar method is also implemented in the Berkeley checkpointing system, having the same constraints.

Due to this reason it is important to add information about the required libraries to the compatibility profile.

In the Linux operating system libraries are saved, having their version in their filename. The library can be found under its major version number due to a link from the real library name to the virtual library name, which only holds the major version in its name (e. g. `libcap.so.1 -> libcap.so.1.10`, or `libnetsnmp.so.5 -> libnetsnmp.so.5.1.2`).

Changes in the patch version usually do not refer to changes in the functions, so that programs running with version `5.1.2` should also restart with `5.1.1`. Minor version changes signal a change in functions, which is backwards compatible to older versions, so that `1.10` should not be restarted with `1.9`.

However, it has to be distinguished between loaded and unloaded libraries at this point. If a library has been loaded, the loader service of the operating system mapped all addresses according to the particular library version. Since this address mapping information is part of the checkpoint dataset, the job would also use the same information at restart.

Even minimal differences in the code of a library has effect on the memory size of that function. This results that address mapping tables are different between two patch versions. If the job restarts with an address mapping table, which is not matching with the installed library, this would cause an incorrect behavior at runtime.

Therefore the checkpointing profile has to distinguish between loaded and unloaded libraries.

- For unloaded libraries it is sufficient to query for a compatible library version.
- For loaded libraries it is mandatory that the identical version is available on the migration target system.

The resource management system can retrieve the library related information about a running job by analyzing the application and checking the list of loaded libraries at checkpoint time.

According to this list of libraries the RMS is then able to check the version numbers of the libraries, adding either the full version or solely the major version number.

If thinking of large Grid systems, having thousands or even millions of providers, it is not practicable to publish information about all libraries installed at each site. Current resource information systems are not sufficiently scalable to cope with such amounts of data.

Therefore the demands on the system libraries is added by the RMS to the SLA-request. The negotiation module at remote site is then verifying if the demanded libraries are available in the correct versions. Only in this case the remote system would be able to agree on the SLA. The source system can assume a high probability that the checkpoint dataset will restart successfully at the remote site, returning correct results.

5 Results and Perspectives

In the previous chapters methods and algorithms have been presented that realize the SLA-awareness in resource management. It utilizes and orchestrates mechanisms provided by the subsystems on process, storage, and network. For further increasing the level of fault tolerance, the system is designed to act as an active Grid component. Instead of solely receiving SLA-bound jobs from local users or Grid users, it harnesses the available Grid infrastructure. By migrating jobs from the local cluster system to other cluster systems in the local administrative domain or even to remote resources in the Grid, the SLA-aware RMS increases the level of service quality.

The described SLA-aware resource management system is not only a theoretical vision. In fact its realization is enmeshed in the objectives of the EC-funded project HPC4U. Thanks to the development partners in that project, sophisticated commercial software products were used as subsystem components, explicitly targeting on requirements and expectations of commercial user communities.

The HPC4U project will end in November 2007. Therefore not all concepts presented in this work have already been implemented and validated. Being in line with the project's working plan, the system is currently able to provide fault tolerance to sequential and MPI-parallel applications. For compensating resource outages the system automatically generates periodic checkpoints of the running applications.

In the case of failures the system is performing an intra-cluster migration according to the requirements of the SLA. This has been validated and presented during the annual reviews of HPC4U in 2005 and 2006. The implementation of cross-border migration mechanisms to other resource management systems within the same local administrative domain is close to completion. The validation of this functionality is planned to be finished until end of 2006.

The next year then will focus on realization of migration to Grid resources. A main focus in this work will be on implementing an interface for the RMS to the Grid middleware.

Numerous other European and non-European projects are focusing the topic of integrating SLA functionality in Grid middleware. By using standard protocols and participating in reference implementation repositories, HPC4U is striving for interoperability with other projects. Concrete agreements have already been consented with other projects during the European Collaboration

Days as well as the Global Grid Forum meetings.

The current state of development of the resource management system using the HPC4U subsystems is installed at the partner sites of University of Linköping, CETIC, and Fujitsu. Furthermore members of the special interest group (SIG) of HPC4U have access to the stable development versions of the software stack. They are evaluating and using the system in its current form, returning feedback on applicability in use-cases and opportunities for improvement. Among the companies in this SIG are Saab Automobiles (Sweden), the Swedish Meteorological and Hydrological Institute (SMHI), the french Commissariat à l'Energie Atomique, the french Electricité de France (EDF), the belgium Centre of Excellence in Aeronautical Research (CENAERO), or the spanish Barcelona Supercomputing Center (BSCC).

5.1 Fault Tolerance Provision

The SLA-aware resource management will enable the Grid middleware to negotiate on service level agreements describing all requirements of the new job. There may be a broad divergence in executed applications, ranging from a small set of standard software packages, up to individually programmed applications. Each of these scenarios may have different requirements on network, storage, or the node environment.

Focal goal of Grid computing is the virtualization of resources. The Grid end-user should be able to consume compute power in a transparent way. After successfully agreeing on a business contract, he submits his job and retrieves results at a given point of time. He does not have to be aware of technical details like file transfer security, access on compute resources or node initialization. In particular this also includes the transparency of all fault tolerance mechanisms.

In this section the functionality of transparent fault tolerance will be described. We selected the padfem application, which is an FEM solver developed at the Paderborn Center for Parallel Computing. It has been selected as example application due to its high demand of resources. At runtime, padfem is using up to several GB of main memory, depending on the selected problem definition. This memory is used for mesh computation and refinement, such that inconsistencies within the internal data structure directly result in a crash of the entire application. Hence, inconsistencies at restarting from a checkpointed state would be detected directly after restart.

Figure 5.1 depicts the submission of a new compute job using the `ccsalloc` command line tool of the CCS resource management system. The padfem application has not been modified or recompiled in this example scenario. Instead, it has been compiled on a different node and then copied.

Beside regular parameters of ccsalloc required in general for all compute job

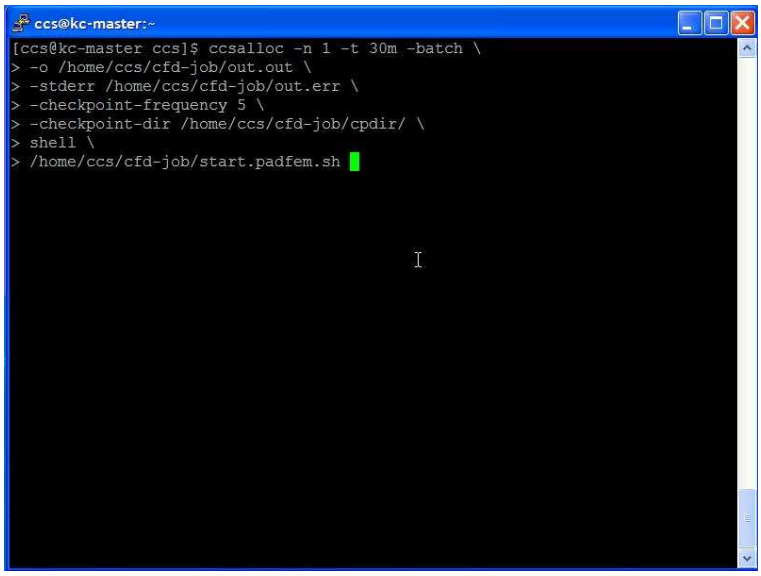

Figure 5.1: Submission of new job

(e. g. `-n` = number of processors, `-t` = estimated runtime, `-o` = file for standard output, `-stderr` = file for stderr output), new command line parameters have been introduced for enabling fault tolerance mode.

The first parameter `-checkpoint-frequency` allows the user to predefine the frequency how often CCS should generate checkpoints for this application. In this scenario, CCS should generate a new checkpoint each 5 minutes. The second parameter `-checkpoint-dir` links to a user provided path for storing new checkpoint datasets. If this parameter is omitted, the default path is used.

The last parameter of the command line specifies a script, which executes the padfem framework for a given problem description. Since padfem requires numerous parameters, the execution has been encapsulated within this script.

According to the command line, the padfem framework should only use one single node for its computation. This can be supervised using the `ccsMon` tool of CCS, which gives an overview about node activity within a cluster system. Figure 5.2 shows that only the node kc1.upb.de is used for computation of the job. There are two bars, representing processor and memory usage.

At checkpoint time, the RMS uses the subsystems as described for generating a checkpoint dataset of the running application. This operation can be observed using the ccsmon tool. At checkpoint time the CPU utilization of the node is low since checkpointing is not a CPU intensive task. After checkpointing, the new checkpoint dataset can be accessed in the specified location (ref. Figure 5.3).

In the case of a node failure, this event is recognized by the internal monitoring mechanisms of CCS. Such an event results in a rescheduling operation, since the

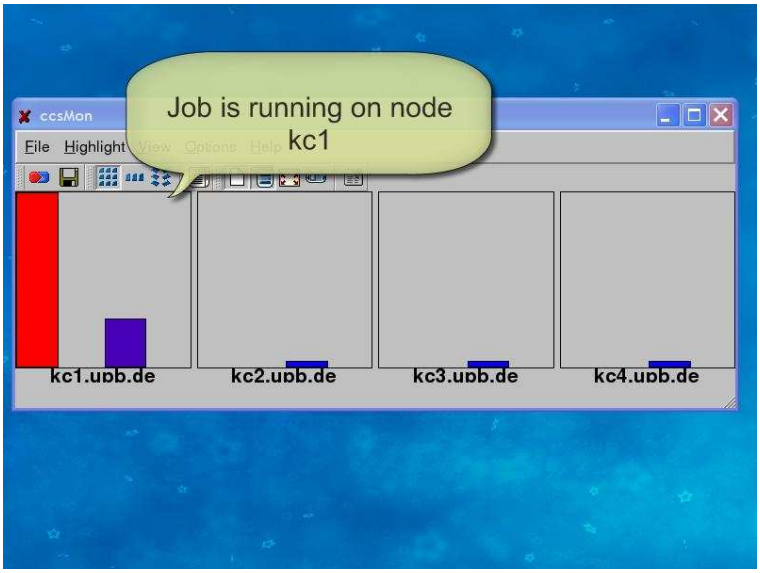

Figure 5.2: Job running on one node

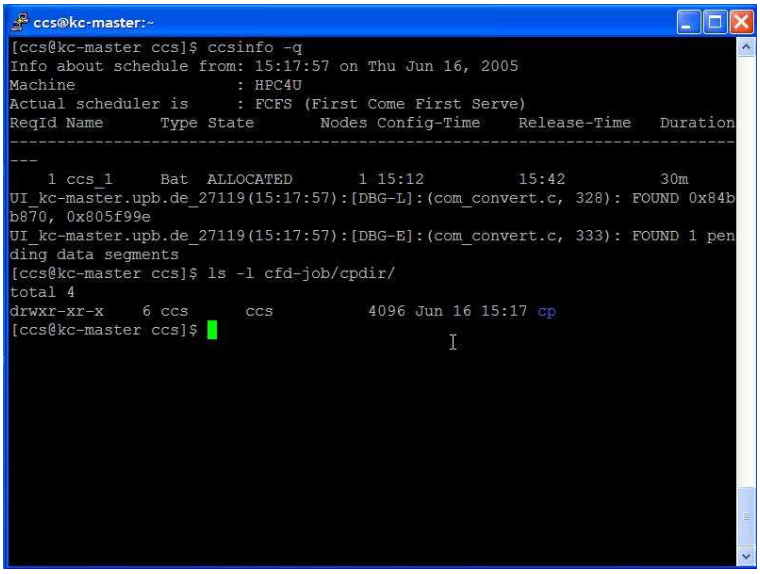

Figure 5.3: Generated checkpoint dataset

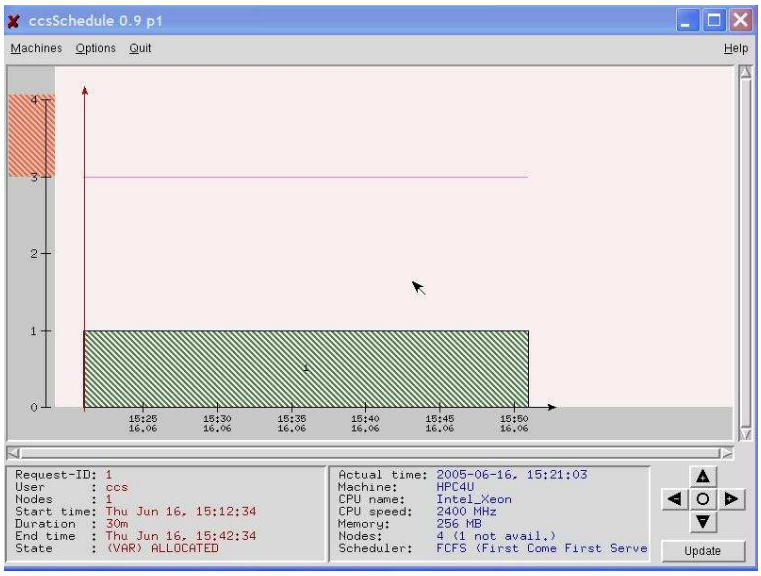

Figure 5.4: System schedule after node failure

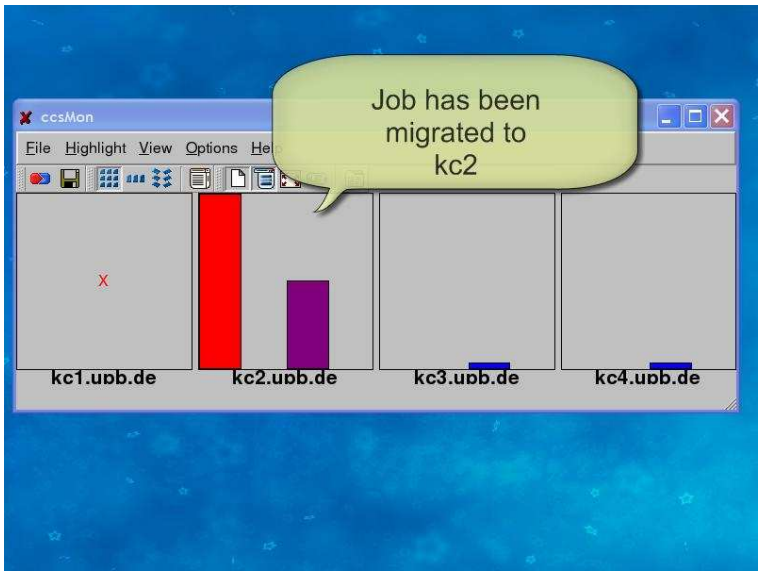

Figure 5.5: Restarted job after node failure

system schedule has to take this new situation into account. In the schedule visualization tools of CCS, this resource failure is directly presented by marking the node and the running job as red, meaning in trouble. However, shortly after this event, the red bar of the job turns green again, because CCS has used the generated checkpoint dataset for restarting the job on a different compute node (ref. Figure 5.4).

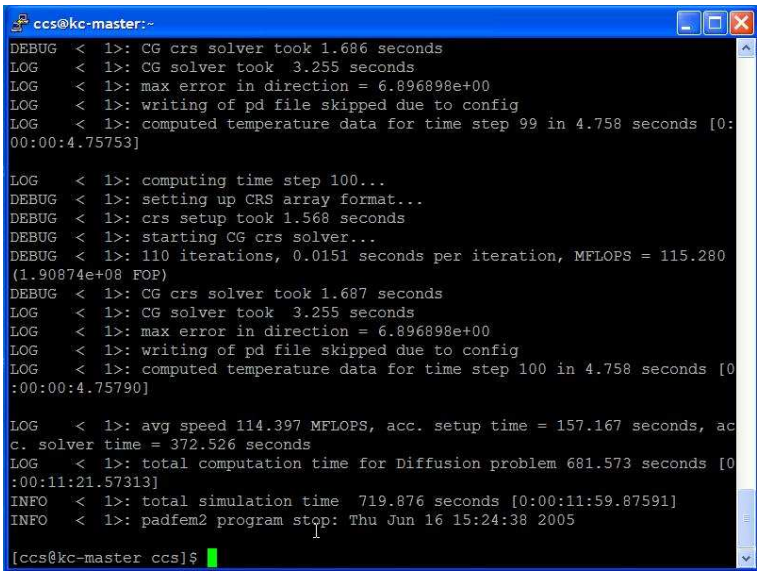

Figure 5.6: Restarted job finished

The node outage is also visualized in the node monitoring tool of CCS. Here, the failed node is marked with an X. The node kc2.upb.de is operating at full load, completing the computation of the affected padfem job (ref. Figure 5.5).

Figure 5.6 depicts the standard output of the padfem framework. According to this output, the computation of 100 compute steps has been completed successfully at 3:24pm, having a computation time of 12 minutes. Since the job has been started at 3:12pm, the node failure had only minimum impact on the finishing time of the compute job, since the result has been only delayed by a few seconds.

5.2 Use-Case Experiences

In this section the experiences of two members of the SIG will be presented. After describing their particular field of work, the improvement coming along with an SLA-aware and fault-tolerant resource management system will be highlighted.

5.2.1 Swedish Meteorological and Hydrological Institute

Hirlam [41] is a multi-national research programme that started 1985. Participating members are meterological institutes in Denmark, Finland, Iceland, Ireland, Netherlands, Norway, Spain, and Sweden. The French meterological institute is a cooperative partner.

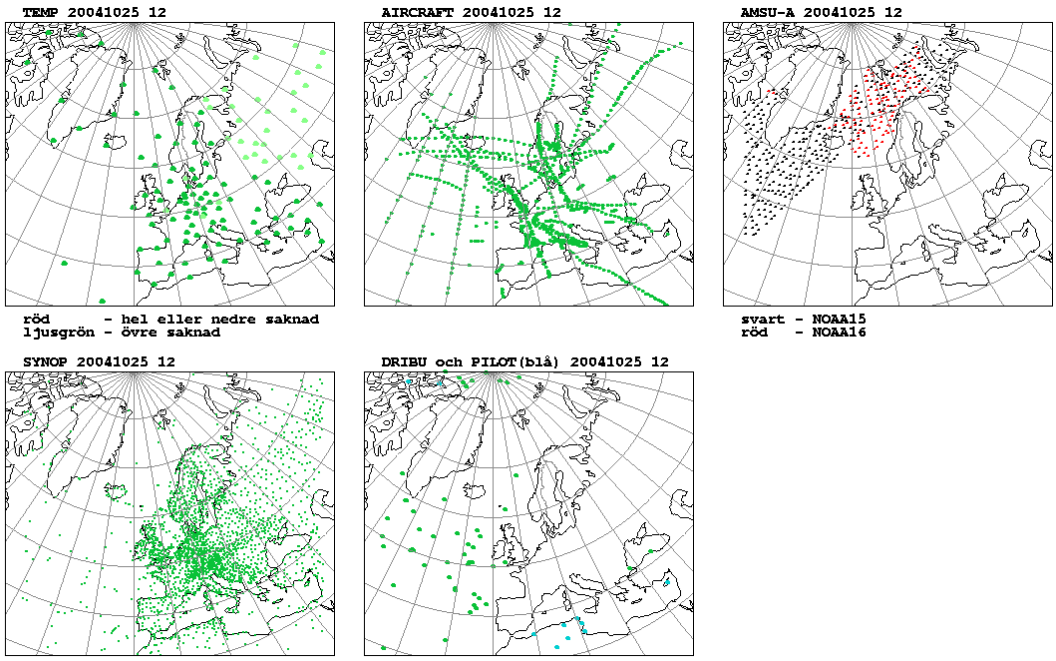

Figure 5.7: Sources for Hirlam Input Data (Image by courtesy of Swedish Meteorological and Hydrological Institute)

The focal goal of this Hirlam programme was the development of a novel numerical forecast model that allows precise short-range weather forecasts. During the last two decades, the Hirlam code has evolved as standard code for regular weather forecasting in Scandinavia and the Netherlands.

In the Hirlam model the initial state is calculated from the current weather situation. Starting from the previous weather situation that was valid at observation time, a first guess is calculated. This first guess then is compared with the initial state calculation, representing the current weather simulation. Depending on the differences between the first guess and the real situation, the code parameters are adjusted, getting a better matching with the current weather simulation. By stabilizing this forecast step, the quality of weather prediction increases.

The main code is called 3Dvar, consisting of a three dimensional variational method. During computation parameters like temperature and humidity in 2 meter height as well as soil humidity are used. These parameters stem from numberless measurement points, respectively adaptive estimates. Other codes use additional parameters like wave and ocean circulation models. Figure 5.7 depicts the various sources for measurements, e. g. observations of temperature, humidity, and wind coming from observation balloons (upper left), civil aircraft

(upper middle), weather satellites (upper right), ground based stations (lower left), or buoys in the sea (lower middle).

The Swedish Meteorological and Hydrological Institute (SMHI) as member of the Hirlam programme also uses this code. Their main calculation is the 48 hours forecast, which has a computation time of approximately 6 hours, having a maximum window of 1 hour. The compute power limits the quality forecast model approximations within the calculation process. A typical run of the 48 hours forecast takes 150MB of input data and generates 3.2GB of output data.

This calculated forecast data is the basis for a variety of weather related products that the SMHI is offering. As an example, SMHI is publishing the regular weather forecast for Sweden to media and end-users.

5.2.1.1 Fault Tolerance Aspects

For calculation of the weather forecast, SMHI is using regular cluster technology. Since clusters are build on standard computer components, single nodes of the cluster are subject to failures at every time. However, the timely completion of computations has to be ensured, even in the case of failures.

Due to the lack of fault tolerance capabilities in current cluster systems, SHMI uses two cluster systems in parallel for executing the forecast. One of these clusters is marked as operational cluster, the other one as backup. In case of no outages, the result dataset generated by the operational cluster is automatically used in the following product generation processes. If outages occurred on the operation cluster, the backup cluster dataset is used instead.

Beside the disadvantage of high administration costs, this type of fault tolerance for SMHI has the main problem in hardware scalability. Since forecasts constantly strive for higher precision, the demands on the computation hardware rise. For SMHI it gets difficult to cope with this, since both cluster systems and their entire environment (e. g. networks, storage, or backup) have to be upgraded in this scenario.

Even if the 48 hours forecast is the main application, SMHI is also executing further simulations and calculations. In the light of cost efficiency it would be beneficial to use existing high performance cluster infrastructure. However, the resource management of this system would have to ensure that these additional tasks do not impact the main application.

As partner of the HPC4U project, the National Supercomputing Center (NSC) at the University of Linköping is closely cooperating with SMHI. NSC operates a cluster for SMHI, where these Hirlam calculations are executed in the HPC4U environment. Compared to standard clusters, this infrastructure underlined its capabilities in successfully coping with resource outages, while holding the requested deadline.

5.2.2 Centre of Excellence in Aeronautical Research

CENAERO is an SME (small/medium enterprise) located in Charleroi, Belgium. It was funded in 2002 as a center for applied research with a strong focus on the development of new simulation methods for the aeronautical domain. Current research at CENAERO focuses on various domains, e. g. virtual manufactoring, damage and fracture mechanics, or flow simulations.

Being located in the same city as CENAERO, the project partner CETIC established the link to a working group at CENAERO focusing on the design of three-dimensional turbomachinery blades. This group is working on establishing an integrated design process starting from the CAD (computer aided design) application, continuing with CFD aerodynamic computations and FEM structural mechanic computations, and finally feeding back simulation data to the design process.

In this context, the optimization of complex shapes is a highly compute intensive task. Various CFD solvers can be used, each offering different optimization algorithms. Here CENAERO is working on an automatic shape optimizer, taking various parameters into account. This solver is called MAX, capable of performing objective optimization.

5.2.2.1 Fault Tolerance Aspects

CENAERO operates its own supercomputing infrastructure, using the compute power to execute simulations using the MAX solver. Depending on the input dataset, a single run can block the entire cluster system for up to two weeks. Due to the nature of MPI-1 parallel applications, even the outage of a single node results in a failure of the entire application.

Even if CENAERO does not have to fulfill deadlines for job completion on the MAX solver, the amount of wasted compute power in case of resource outages is immense. Moreover the failure of a long running simulation delays the development process, impacting the efficiency of the employees working in the development group.

In the cooperation between CENAERO and HPC4U, cluster systems of CETIC are used as evaluation platform. The MAX solver application is running in fault tolerant mode without any modifications necessary on the binary. In case of resource outages the computation does not restart from scratch, but the MAX solver resumes as planned from the latest checkpointed state.

5.3 Key Requirement Security

Service level agreements are contractual agreements, defining all expectations and obligations in the business relationship between service consumer and ser-

vice provider. Integrating service level agreement awareness to resource management systems is inevitable when commercializing Grid infrastructures.

Having this as core idea of this work and central objective of the HPC4U project, dissemination of architectures and achievements started. While talking to potential users it became obvious that the understanding of the terms of a contractual agreement exceeded the initial considerations. This impression was underlined by discussions held with representatives of the special interest group of the HPC4U project.

Beside the envisaged goals of fault tolerance, deadline compliance, and service quality adherence, security was of focal interest. These security demands exceeded the expected scope of access control or encryption of data transfers.

In particular providers raised the issue that they have to ensure a high level of security to their customers. This does not only include the establishment of firewall mechanisms separating the internal computing infrastructure from the public Internet. Security mechanisms are also required within the compute center, preventing malicious customer jobs from causing any damage.

This demand of establishing security contexts for executing jobs within cluster systems is new in Grid computing as well as resource management community. However, it is a central demand of many commercial stakeholders. If commercial providers should be attracted to provide their resources to their customers by means of Grid technology, both partners have to be available to negotiate also on these aspects.

For the negotiation process this novel requirement can be solved quite easily by introducing new service description terms. At level of resource management this demands novel mechanisms, allowing to establish intra-cluster security mechanisms as well as their surveillance at runtime.

Firewalls will be a major instrument in realizing inner system security. By establishing micro-firewalls on the nodes of the cluster system, the resource management system is able to partition the hosts according to their job assignments. In case of malicious jobs running in one partition, these jobs would have no opportunity to contact nodes of the cluster not belonging to their partition.

Customers will be able to request the execution of their jobs within these restricted environments by adding this as parameter in the negotiation process. At runtime the RMS is in charge of configuring the micro-firewalls on all nodes accordingly.

Additional security can be achieved by installing intrusion detection systems (IDS) within the cluster. These software systems are analyzing network traffic, trying to detect unexpected data flows. This mechanism further increases the security level of micro-firewalls, since attacks may be detected by the IDS, even if they successfully passed the activated micro-firewall.

Kernel hardening further increases the level of security. Beside removing unnecessary software packages or protection of daemons by means of network

transport protocols, this also includes the usage of specifically patched kernels (e. g. introduction of user stack areas for non-executable code only, which makes it more difficult to get root privileges by provoking buffer overflows). As a matter of fact kernel hardening is no system service which can be enabled. However, the resource management system could boot the compute node with a hardened kernel prior job execution. This way suspicious jobs (e. g. submitted by untrusted Grid users) may be executed in a specifically secured environment.

Beside kernel hardening sandboxing is another way for securing the system against malicious code. Instead of executing the job in a vulnerable environment, it is executed in a specific partition of the cluster system, which is not physically connected to the other compute nodes. Malicious jobs could only attack other jobs running in this sandbox area of the system, but no valuable jobs of important customers. Sandboxes can also be realized by means of virtual machines, virtualizing the execution environment for the started questionable job. If this job wants to attack a job running on another node, it first has to succeed in escaping from his virtual environment.

If a Grid technology and Grid-enabled resource management system really should enter the domain of commercial stakeholders, security functions like the mentioned mechanisms have to be provided. Moreover customers could ask for execution at security certified providers, which have to follow regulations of these certificate programs. Here, providers would not only have to establish security mechanisms at their local site, they also have to prove the appliance in a given form.

Currently a diploma thesis [71] is focusing the introduction of security mechanisms in the RMS domain. Beside the analysis of available methods and tools, this work will also present a prototypic implementation. For attracting additional user communities for the idea of SLA-aware resource management, these achievements will then also be implemented within the scope of the project.

6 Related Work

The development of SLA-aware resource management aims at closing the gap between functionality demands of Grid middleware and capabilities of current resource management systems. Commercial users demand their Grid infrastructure for contractually fixed service quality levels. At the bottom line these demands have to realized by means of local RMS, which are offering their resources to the Grid infrastructure. By introducing SLA-awareness to these RMS, the Grid middleware is enabled to base their binding promises to the customer on binding promises given by providers.

Numerous research groups and projects worldwide focus on introducing SLAs in Grid Middleware. This chapter will highlight major developments of both ongoing and already concluded developments. It will also point out intersections, distinctions, and connecting points to SLA-aware resource management presented in this work.

Checkpointing and migration are important building blocks for realizing SLA-compliance in case of resource outages. This chapter will also present developments within the domain of resource management focusing on these aspects. Furthermore this chapter will also address base technologies required in the subsystems of this SLA-aware resource management system.

6.1 Service Level Agreements in Grid Middleware

In [35], 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.

6.1.1 Advance Reservations and GARA

Traditionally resource management systems are queuing based. Having one queue or multiple queues with different priorities, new jobs are assigned into such a queue. Resources of the cluster then are assigned to jobs in the front of these queues. Due to the fact that the user does not have to specify the runtime of his job, the system does not now the finishing time, nor the time when an

arbitrary job within a queue will come to execution. The targeted field of usage of these system clearly was high troughput computing (HTC), not the provision of service guarantees or realization of reservations.

For ensuring the availability of (compute) resources at a given point of time, advance reservations have been introduced. Here, the user is not only able to specify the amount of required resources, but also the required time frame. Advance reservations first have been introduced by means of high priority queues, bringing the particular job in front of that queue directly to execution.

The Maui scheduler [68] is not a resource management system itself, but a plug-in scheduler component which can be integrated into several other resource management systems (e. g. Loadleveler [51], OpenPBS [83], or Sun Grid Engine [92]). Maui does not replace core system components, but enhances the already existing scheduler by new features like job priorities or configurable node allocation and backfilling policies [17]. Since the concept of reservations has also been introduced to Maui, reservations are available in a number of resource management systems after installation of the Maui scheduler.

Development on the Globus Toolkit began in the early days of Grid computing (ref. Section 2.2). Already at that time the demand was recognized for guaranteed service provision, e. g. the guaranteed availability of compute resources at a specific point of time, so that the orchestrated operation of workflows can be realized.

Another targeted field of application was the multi-site execution of parallel application. Here the execution of an MPI-parallel job is split to resources of different Grid sites (e. g. execution on nodes of different clusters systems). Prerequisite for making multi-site execution work is the simultaneous availability of compute resources, so that the multi-site distributed instances of the MPI-parallel job could start in parallel.

Since providers invest remarkable amount of budget in highly sophisticated network technology bringing high capacity and low latency, the performance of multi-site jobs using standard LAN or even WAN connections was unsurprisingly poor. This type of parallelism only is reasonable in cases where problems are to be computed not fitting on a single parallel machine (e. g. due to required number of nodes or available main memory), so that performance is not the decisive factor. Hence multi-site application never became mainstream usage.

In the late 90s the development on an architecture began which supported the co-allocation of multiple resource types within the Globus toolkit, e. g. processors on compute clusters or network bandwidth. The Globus Architecture for Reservation and Allocation (GARA) was first presented 1999 in [50]. The designers of GARA were aware of the fact that only few resource management system (e. g. for compute resources or networks) were able to provide reservation mechanisms.

To rise acceptance of GARA in a non-reservation environment, "wrapper"

functions were introduced [28, 88]. First, these functions unified the access to resource managers. This way upper level layers do not have to know about calling details of different resource managers. Second, these wrapper functions tried to emulate missing functionality as good as possible. This way it was possible in some cases to provide reservation service even if the underlying resource manager does not provide it originally.

In GARA reservations were specified using RSL [89] (resource specification language). The user had to specify his reservation demands in form of an RSL coded request. If GARA was able to comply with this request, it returned a handle to the application, which then could be used to consume the reserved resources. Beside creation and deletion, GARA also supported the modification of agreed reservations, given that the modified reservation could be realized on the requested resources. At runtime GARA allowed the monitoring of reservations using both polling (i. e. user actively requests for information about the current state) and pushing (i. e. user specifies a callback function, which is then called by GARA for submitting information about the current state).

GARA has been an important step towards an integrated QoS aware resource management, setting the first step in introducing SLA-awareness in Grid middleware. However, GARA had to deal with the same limitations as current SLA-middleware components. For realizing reservations it has to rely on failure free operation in service providing resource management systems. Moreover it does not support internal resilience mechanisms to handle resource outages or failures.

6.1.2 Service Negotiation and Allocation Protocol

The next important step in introducing guaranteed service provision into Grid middleware was done with the Service Negotiation and Allocation Protocol (SNAP) [62]. It describes the requirements and procedures of a protocol for negotiating SLAs within Grid middleware, in particularly focusing on the multi-phase nature of this negotiation process: beginning with service requests from the user and the discovery of appropriate providers, followed by negotiation and reservation, up to configuration, monitoring, or re-negotiation. SNAP provides a model allowing to perform these steps.

SNAP addressed the fundamental problem of competing interests of service consumer and service provider within the service negotiation process. On the one hand the service consumer is interested in a maximum level of understanding and insight. On the other hand the resource provider aims at retaining a maximum level of autonomy and control over the resources in his administrative domain. Even if the provider will not undisclose all information about his resources, the user (or an automated system at Grid middleware, like a Grid broker service) has to be able to determine if a resource is appropriate for a

given job and what performance and service characteristics can be anticipated.

For this purpose SNAP proposes the introduction of three different kinds of service level agreements:

- Resource Acquisition Agreement (R-SLA): These SLAs are solely agreements regarding the right to consume a specific resource in a given manner.
- Task Submission Agreement (T-SLA): By means of this SLA a resource can be informed about an upcoming task.
- Task/Resource Binding Agreement (B-SLA): Bringing together an agreed R-SLA with an agreed T-SLA. This way a given resource can be used for a given task.

The SNAP protocol decouples the negotiation process, allowing the distinction between resource related parts and task related. This is important if tasks can only be provided in a specific resource environment. Resource providers in this case can only agree on providing a task if the resource environment is known in which this task is to be performed.

SNAP raises a general problem on SLA negotiation in Grid middleware. By distinguishing between R-SLA, T-SLA, and B-SLA the middleware components are able to first all required hardware components, then to check if tasks can be mapped in this environment, and to finally assign tasks to the resources.

These basic considerations did not only enter the development of the protocols WS-Agreement and WS-AgreementNegitiation within the GRAAP working group of the Global Grid Forum (ref. Section 2.3.1.3). It primarily impacted the development of OGSI Agreement [60], a mechanism at level of Grid middleware to create OGSI compliant Agreement Services.

However, the problem of SLA mapping on dynamic composition is affecting more the workflow broker at Grid middleware as the submission of a single task. Therefore the impact for the local resource manager is limited, because this system is always able to decide on the contents of a single negotiation request.

The resource manager developed in this work only supports a subset of the state defined in Section 2.3.3. In particular, it does not support re-negotiation, which would be vital for mapping a task to an environment that has not been specified at initial negotiation time. The focus of this work is on reliability and fault-tolerance. It gives an answer how a local resource provider can adhere to the contents of an agreement, in case his local resources do not operate as planned. This aspect of fault tolerance or even failure handling and recovery has not been handled by SNAP.

6.1.3 Grid Broker Services

Compared to advance reservations, SLAs offer a greater scale of freedom. Instead of finding available time slots for the resources at provider level, a more flexible SLA can be negotiated, e. g. defining a deadline as latest possible time for completion of a given task in a branch. Hence, broker services in particular benefit from SLA technology in Grid middleware level. Grid brokers are responsible for mapping a workflow to available Grid resources. These workflows can either be defined by the user in his service request, or selected by the user from the broker's repository of known workflows.

Broker services can also perform the task of Grid level schedulers (also known as superschedulers). These components are responsible for acting as matchmakers between service customer requests and available resources. According to internal scheduling policies they decide which job will be executed by which provider.

Research on scheduler services and Grid broker services is focused in numerous research projects. Selected projects of particular importance will be highlighted in the following.

6.1.3.1 GRUBER/DI-GRUBER SLA broker

The development on the GRUBER broker system [9] started at the Computer Science Department of the University of Chicago and the Argonne National Laboratory. This location underlines the close relationship with the Globus Toolkit. GRUBER allows the automatic selection of resources that are available within a virtual organization, following restrictions and rules specified in the virtual organization's policy.

Within the GRUBER approach a job is characterized by the properties of VO-membership [48, 99], group (defining the execution context), required processor time, and required storage space. The broker now has to find an assignment that satisfies all policies and optimizes regarding provider/VO utilization or other objectives.

The GRUBER system consists of four main components. The engine encompasses all algorithms necessary for determining optimized resource assignments. The site monitor is acting as data provider, publishing selective numbers about the local resource system to the Globus environment. The site selector is responsible for selecting a resource provider for execution of new tasks. The algorithms of this selection process are part of the engine, so that these two components are communicating. The queue manager is installed on submitting hosts, deciding which jobs can be executed at what time.

GRUBER has been implemented in Globus Toolkit versions 3 and 4, both for web-service and pre-web-service flavors. It is promoted as part of the official

Globus toolkit family, e. g. at Supercomputing 2005 conference, and therefore has significant impact in the Grid broker domain. The DI-GRUBER [10] architecture is an enhancement of the GRUBER system, allowing it to work in a larger scale Grid. In contrast to GRUBER, DI-GRUBER has no single point within the system for taking scheduling decisions. Instead, schedulers are working independently, exchanging information only rarely and in a loosely coupled fashion.

Even if GRUBER is called to be an SLA broker service, the notion of an SLA differs significantly from the ideas of this work. Here, an SLA does not cover any service quality parameters, but only basic job profile information. This is due to the fact that GRUBER has been developed for the classic Globus environment, which is using queuing based resource management systems only for realizing jobs. For GRUBER the specification of processor time and storage space is essential to find an optimal match to those systems.

This origin is also underlined by the architecture of the system. Instead of negotiating on these systems, GRUBER demands special services running on these machines, publishing data like utilization to the central GRUBER component. This central component then takes the scheduling decision, without the option of interference from the local RMS. Commercial providers in particular strive for obtaining their local autonomy when providing their resources to Grid systems. Therefore it is doubtful if this approach will be accepted here.

6.1.3.2 White Rose Grid

The White Rose Grid (WRG) [102] is a center of excellence, funded by the British eScience initiative. WRG started in 2002 with the universities of Leeds, Sheffield, and York as providing a Grid infrastructure. It is accessible via Globus Toolkit and provides various services on computation, e. g. data storage or local compute clusters. Furthermore WRG provides access to selected applications over dedicated Grid portals. This infrastructure is used as platform for various other projects in the eScience initiative.

With a rising number of resources, applications, and users the need for a broker service increased. The realization of this broker service [24] also enabled the execution of workflows on the WRG, since the broker service was responsible to find and allocate suitable resources for each workflow step. It tries to match the requirements of each particular workflow step with the profile of available resources. This broker supports the negotiation of SLAs both between user/broker and broker/resource. It implements the SNAP protocol (cf. Section 6.1.2), where the T-SLA represents the user's specification of the task requirements, R-SLA represents the discovered and allocated resources, and B-SLA the assignment of a task to a resource.

For realizing tasks on compute resources, the WRG broker service has to uti-

lize local resource management systems, providing their resources to the WRG Grid infrastructure [57, 56]. Since the WRG aims at providing a higher service quality to the upper layers, a gap between the provider capabilities and the broker demands becomes apparent.

This problem even reveals during the allocation process. If matching multiple tasks to multiple resources, the broker asks the Grid information service for providers offering appropriate resources, followed by the direct request at these providers for the current state of these resources. Since the provider does not provide SLA negotiation mechanisms, the broker can not be sure that the discovered resources are still free at a later point of time in the internal matchmaking process. In that case the broker would have to repeat the entire matchmaking process, hoping that resources remain free until the end of the matchmaking process this time. This problem is answered by starting processes on the particular provider machines, directly notifying the broker about availability changes. Beside these basic negotiation problems, this architecture also has to rely on the reliability of local systems, not offering any fault tolerance.

The results from this work would perfectly complement the functionality of the WRG broker. In fact, an active cooperation with the University of Leeds has already started in the scope of the EC-funded project AssessGrid [4]. Having an interface available offering the standard WS-AgreementNegotiation protocol, the WRG broker as regular Grid middleware component may directly start negotiations on new SLAs with the SLA-aware cluster.

6.1.3.3 NextGRID

NextGRID [74] is an EC-funded project, started in the same call as the HPC4U project (cf. Section 4.1). The focus of NextGRID is to work on tomorrow's Grid systems. Following the demands of the Next Generation Grid reports [34, 35, 59] this explicitly includes commercially successful Grids and the evolvement of new business models. Hence, the introduction of service level agreement technology into various Grid architecture elements is of central importance.

As novel key element NextGRID will introduce partnership SLAs (pSLAs), spanning over the entire lifecycle of a commercial business service: starting at negotiation time, followed by deployment, execution, and monitoring, up to accounting and decommission. Even if the service is a leading principle here, the basic idea of a pSLA equals a regular SLA: a contractual agreement of all obligations and expectations between provider and consumer.

The NextGRID project will also work on realizing broker services, executing services by means of locally available resources at provider level. Even if no implementations of this broker service exists yet, this broker will have similar limitations like the WRG broker. These limitations are given by the nature of non-SLA aware resource management systems used for executing jobs.

In the scope of European collaboration, NextGRID is following the achievements of HPC4U. Presuming the interoperability of NextGRID's SLA approach with common standards, SLA-aware resource management will also allow the NextGRID broker to negotiate on service levels with local resource providers.

6.1.3.4 SLBSH Project

The focus of the British EPSRC funded project Service Level Based Scheduling Heuristics (SLBSH) is to evaluate the impact of SLA based system management.

In the scope of this project, Yarmolenko et al presented the results of a parameter study in [98]. Due to the lack of SLA infrastructure and components, they implemented a simulation environment assuming a central scheduler instance (in this paper called coordinator, acting as the one and only connecting component between incoming requests and available resources. In this scope, resource requests only differ in the four available service level objectives (earliest start, deadline, runtime, number resources).

Even if this paper has interesting results regarding the efficiency of scheduling algorithms in specific load scenarios, it assumes a simplified reality. Real world will neither work with a central scheduler component, nor with unified resources or four service level objectives only. Moreover this work assumes that the local resource provider will accept all incoming jobs until his schedule is utilized (which is known in advance by the coordinator by taking the difference between submitted jobs and available resources). It neglects the existence of local users directly submitting jobs to their system, as well as schedulers in local RMS aiming at different goals, e. g. not optimizing for utilization but revenue.

Furthermore the failure aspect has not been regarded in this work. Instead of expecting failures at local resource level - the driving assumption of this work - , all jobs are assumed to be executed as agreed. Thus it does not regard the impact of these events to the schedule quality at Grid middleware level.

Having this work implemented and used in real practice, resource failure profiles for these simulations may be extracted from logfiles and scheduler traces. In general results from this work could be beneficial in general if thinking about understanding of the work of an SLA-aware resource management system.

This statement also holds valid for earlier work of this group, where Yarmolenko et al evaluated policies for negotiation with resources [98]. The paper does not focus on local resource management systems but broker services only. The realizability of these policies is questionable, since the local resource management system only has full control over the resources. Only if this RMS agrees, upper layer instances may have impact on questions like fault tolerance and QoS provision.

6.1.4 Grid Economy

Introducing service level agreements on all levels of the Grid - including the local resource management systems - is only just the first step towards a commercial Grid system. Business customers do not only request for contractually fixed levels of service quality. They also demand for traceability and verifiability.

The crucial question in this context is, whether the provider complied to all requirements of the contract. The current approach of trusting the statement of the resource provider obviously is not sufficient in business critical commercial environments, where non-compliance results in severe penalty fees. Both parties therefore have reasons for distrusting each other. The customer may accuse the provider for violating terms of the SLA (e. g. providing insufficient network bandwidth at runtime), but reporting SLA compliant operation. The provider in contrast may refer to incomplete service requests received by the customer or other external reasons for job failure, to prevent paying any penalty fees.

SLAs therefore do not only have to be negotiated and regarded in system management, their fulfillment also has to be monitored. In the optimal case this monitoring is performed by neutral third party instances that both contractual parties can trust. Already in 2002 Sahai et al [2] addressed this problem, proposing the introduction of an SLA monitoring engine at Grid middleware level. This engine is fed with information from manageability interfaces of the resources or other agents providing measurement data (e. g. SNMP agents for network related information).

This problem is also targeted by the Brazil OurGrid project [84], having Hewlett Packard Brazil as commercial partner. Service level indicators can be used to assess the service properties of a provider, but how may an SLA auditor service determine if reported service level indicators are trustful? In [3] an architecture is proposed, introducing an inspector service and auditor service, which both parties are trusting. The inspector service is interposed between the provider and customer. Instead of directly accessing the service provider's interfaces, the customer accesses the service inspector, which masquerades the provider's service. This way the inspector can monitor all provided services and report measurement information to an auditor service. This service then is able to decide on SLA-compliant service fulfillment.

Presuming the trustworthy decidability of SLA-fulfillment (and therefore the accepted enforcement of penalty fee payments), models on Grid economy have to be developed. These models are representing novel business models which should evolve in future commercial Grid systems. In [87] Buyya et al give an overview about these Grid economic aspects. In this scope new Grid services will be required, like Grid Trade Server (managing the selling process of resources from providers to customers, applying pricing policies and ensuring proper accounting) or Trade Managers (discovering resources available in the Grid that

match the customer's demands and optimize the costs).

6.2 Process Checkpointing

Process checkpointing is a well known technique for creating an image of a running process. Various checkpointing solutions exist, ranging from application driven checkpointing operations up to kernel level solutions. A focal goal of the HPC4U project is to provide fault tolerance to the application in a transparent and non-intrusive manner. Thus, the HPC4U has to be able to provide checkpointing services to the running application without the need of changing or recompiling the application. For this reason, most available checkpointing techniques are inapplicable within the project context.

Application level checkpointing solutions realize checkpointing and restart by dedicated code segments within the application. Even if this is highly efficient (due to the fact that only required information is written to the checkpoint dataset file), it implies additional programming effort for the application provider. If an application does not support checkpointing, the user of this application has no opportunity to realize it on his own.

In contrast, user level or kernel level approaches do not require the application source code to be changed. Where user level approaches provide specific checkpointing libraries, kernel level approaches modify or enhance the operating system kernel.

Most user level checkpointing solutions are realized as special purpose libraries, which then are linked against an application. For this, the application in question has to be available in source code or at least in object code. In contrast to kernel level checkpointing, user level checkpointing does not need the kernel to be changed. At runtime the linked checkpointing libraries intercept a certain set of system calls coming from the application. This way, the checkpoint system can monitor the current application state and properties.

The Condor checkpointing solution [12] is a well known and widely used representative for user level checkpointing solutions. Presuming that the Condor checkpoint libraries have been linked against the application, the generation of the checkpoint then can be initiated by sending a signal to the running application. This signal then is caught by routines in the checkpointing library. Even if Condor does not require any modifications of the application itself, the programmer of an application has the opportunity to actively support the checkpointing system. Using checkpoint library provided functions, the programmer can for instance mark specific parts of functions in his application as not interruptible.

Beside the necessity of linking additional libraries against the application, user level checkpointing approaches have additional drawbacks, like the inability of accessing kernel level information about the process, or limited support of

process groups. These systems also have limited capabilities for restart and migration, since they cannot modify all application properties, like the ID of the running process.

Particularly due to the necessity of linking additional libraries against the application, these approaches are not applicable within HPC4U. The goal of application transparency implies that any kind of application may benefit from fault tolerance mechanisms, not only those where source or object code is available. The user does not even have to know about the system's fault tolerance abilities. Therefore, HPC4U solely focuses on kernel level approaches on realizing checkpointing.

The HPC4U project will have two vertically integrated systems as project outcomes. The commercial outcome will consist of commercial components for the storage, process, and network subsystems, whereas the non-commercial outcome will consist of non-commercial and open-source components only. Hence, both outcomes will base on different systems for process checkpointing. The commercial outcome will use the Metacluster system of IBM, the freeware version will use Berkeley Labs Checkpointing and Restart (BLCR). Both systems differ in their functionality and will be presented in the following.

It is vital that the target platform is compatible to the failing resource. If both platforms are incompatible (e.g. if libraries are not available at target node), the application will crash at restart. Therefore, the process subsystem will generate a requirement profile of each checkpointed job. This profile will be used by the resource management system for finding compatible resources, which may be located on the same cluster system, a different cluster system within the same administrative domain, or even somewhere else in the Grid.

6.2.1 Berkeley Labs Checkpointing and Restart (BLCR)

The BLCR system has been developed by the Future Technologies Group of Berkeley Labs. Even if BLCR does kernel level checkpointing, it does not require the kernel to be patched. All checkpointing functionalities are realized by means of kernel modules which have to be compiled specifically for the used kernel configuration. The modules then can be loaded without the necessity of rebooting the machine.

BLCR both supports Linux kernel 2.4 and kernel 2.6 and a large range of processor architectures, including 64bit processors like AMD Opteron. BLCR ships with the powerful feature of callbacks to user-level code. These callbacks are triggered by BLCR if the system is about to generate a new checkpoint. If an application uses these callbacks, it is able to release all critical resources, i.e. which can not be checkpointed or released by BLCR. Examples for those resources are open network connections and open files. After reception of this

callback, the application can close these resources, re-opening them after checkpoint has finished.

The BLCR system does not support the checkpoint of parallel applications by itself. However, systems like LAM-MPI are providing the checkpoint of MPI-parallel jobs by using the callback feature of BLCR. The user does not have to modify his application for benefiting from this checkpoint capabilities, but only to re-link against the MPI libraries of LAM.

While checkpointing, BLCR creates a file for each checkpointed process, named context.pid, where pid is the process ID of that process. On checkpointing parallel jobs, for each process and node a separate file is written, plus a separate file for the mpirun process. The size of the checkpoint dataset approximately equals the sum of memory allocated by the checkpointed application.

BLCR is used for the non-commercial outcome of HPC4U, since it is open source and can be used free of charge even in commercial environments. Our tests showed that it can be used with a large variety of applications, so that the interested user can get a good first impression of the HCP4U system. However, the system has some drawbacks compared to the commercial outcome, which mainly affects the migration of processes to other machines. First, BLCR does not virtualize the system environment. Hence, it restores all job parameters on restart, e.g. process IDs. If the process ID is already used by another process, the restart fails. The same holds valid if the application relies on node specific properties, which differ on the node used as migration target. On checkpointing MPI-parallel applications using LAM-MPI, the name of the source nodes is stated in the checkpoint file. It is necessary to change this nodename to the name of the target host for successful restart.

6.2.2 IBM Metacluster

Metacluster is a product of the HPC4U partner Meiosys, which has been recently acquired by IBM. Metacluster is also a kernel level checkpointing solution. In contrast to BLCR it requires the Linux kernel to be patched.

If an application should be checkpointed at runtime, it needs to be started using Metacluster commands. This way, the Metacluster system will not only start the application, but start it within a “virtual bubble” around the process, presenting a virtual environment to the running process, e.g. consisting of virtual network devices or virtual process IDs. The impact on the runtime of the application due to the virtualization overhead is marginal and below one percent. If a checkpoint has to be generated, the entire virtual bubble is checkpointed. Since no recompilation or relinking of the application is necessary, this checkpointing process is transparent for the running application.

The crucial benefit of running the application within a virtual bubble becomes apparent at migration. After the checkpoint dataset has been transferred

to the target host, the Metacluster system resumes the job from the state of the checkpoint dataset. For this, Metacluster again establishes a new virtual bubble, where the application executes. Due to the virtualization of the environment, the resumed application will not notice that it has been relocated to a different machine. In case of network communication between multiple virtual bubbles, Metacluster wraps the relocation of the jobs to new target machines, so that jobs automatically communicate to the correct new target host. On communicating with hosts outside the Metacluster system, this relocation can not be provided, since Metacluster can only impact on virtual bubbles provided by the system. However, Metacluster provides relocators for network traffic, which fetch incoming network connections, relocating them to the new target host, where the application processes the input.

HPC4U explicitly focuses not only on fault tolerance support for single node applications, but also for parallel applications. Checkpointing mpi-parallel applications has additional demands on the checkpointing system, since it requires the orchestrated checkpoint operation on all nodes as well as the checkpoint of the network state (e.g. network stacks and in-transit packets currently transmitted over the network). For this purpose, the HPC4U partners IBM and Scali created the Cooperative Checkpointing Protocol (CCP), which ensures the fulfillment of these goals. At checkpoint time, Metacluster directly interacts with the Scali MPI.

While checkpointing, Metacluster creates the checkpoint dataset consisting of multiple files in a user specified directory. On checkpointing MPI-parallel applications, this checkpoint dataset contains checkpointed process data from all nodes as well as the network checkpoint. Similar to BLCR, the size of the checkpoint dataset approximately equals the sum of memory allocated by the checkpointed application.

7 Conclusion

If commercial users should be attracted to use Grid environments for computing their business critical and deadline bound jobs, Grids have to be able to provide contractually fixed service quality levels. This demand also affects local resource management systems, which are providing their resources to these Grid systems. Grid middleware components can only agree on contractually fixed QoS levels, if these agreed services can be realized by means of local resource providers.

Currently resource management systems are operating on the best-effort approach only. For realizing SLA-awareness in these systems, numerous components are essential. Beside integrating mechanisms to enable the negotiation of new SLAs with Grid middleware components, SLA-awareness primarily has to be integrated in the system management process. Providers have to pay penalty fees as agreed within the SLA contract. Hence, providers have to rely on their resource management systems, delivering service with respect to all agreed QoS demands.

SLA-awareness in particular implies the consideration of resource outages as normal events in system management. Resource management systems operating business critical jobs have to be able to cope with resource outages or other unforeseen system behavior. Fault tolerance mechanisms have to be integrated, providing their service to the running application in a transparent manner. This transparency is crucial since the main task of Grid middleware is the virtualization of Grid resources. Moreover this transparency is crucial for supporting commercial applications, where recompiling or relinking is not possible.

In this thesis an approach for realizing SLA-awareness in resource management systems is presented, which complies to these demands. It provides an interface allowing Grid middleware components to negotiate on new service level agreements. The RMS only accepts new SLA-requests if all terms can be fulfilled according to the current schedule and system condition. At runtime the system uses its mechanisms to ensure the SLA compliance. For providing transparent fault tolerance and consistency at restart, it furthermore uses the services of subsystems for storage, process, and network.

For further increasing the level of fault tolerance, we introduced the idea of cross-border migration, where resources on other cluster systems are used for restarting a checkpointed job. If the resource management system is embedded within a Grid infrastructure, the system can even try to migrate to other Grid resources. This significantly increases the number of potential migration

target resources. Potential resources must comply to all terms of the SLA of the job to be migrated. Moreover the resource has to be compatible to its latest checkpoint dataset. For verifying this aspect, a compatibility profile has been proposed. This profile covers all aspects deciding the compatibility,e. g. hardware properties or software library versions.

The implementation and realization of SLA-aware resource management has been carried out in the scope of the EC-funded project HPC4U. With IBM, Seanodes, Dolphin, and SCALI as commercial partners focusing on subsystem functionality, it was possible to realize a vertically integrated system, ready to be used in commercial and productive environments. Early experiences coming from use-case experiments in the scope of the project confirmed the applicability of transparent fault tolerance and migration mechanisms.

Discussions with potential users of SLA-aware resource management systems also revealed the high importance of security demands. Beside general security mechanisms, providers ask for methods to establish security levels within the cluster system, e. g. isolating compute nodes by means of micro firewalls or sandboxing functionalities. These security aspects have to be introduced as negotiable QoS parameters as a prerequisite for broad commercial acceptance.

The focal goal of SLA-aware RMS is the adherence with all agreed SLAs. In case of resource outages the number of spare resources is the limiting factor for compensating the outage. The ability to migrate to cluster-external resources significantly increases the RMS' options, because it increases the number of potential migration targets. While the compatibility between cluster systems within the same administrative domain can be configured statically, this is not possible for the migration to Grid resources. Here, information from the compatibility profile can be used to find compatible target resources.

However, even if the compatibility profile ensures the successful restart of the checkpointed job on the migration target resource, Grid migration is not promising in today's Grid infrastructures. Firstly, the size of checkpoint datasets can be huge, since a dataset also contains the storage contents of an application. This problem currently can only be tackled by background replication of storage contents at job runtime. As a matter of fact, this requires the precautionary reservation of backup resources. Having network bandwidth steadily rising, on demand migration of large datasets will become possible in the future.

The second problem relates to currently existing Grid infrastructures. Even the EGEE [21] testbed as the largest currently existing Grid deployments only encompasses approximately 100 resource providers with a total of 50000 CPUs. The compatibility profile puts heavy demands on the potential target resource. Even if all resource providers in this testbed were able to negotiate on SLAs, the probability of finding a matching resource agreeing on all terms of the SLA (in particular regarding QoS constraints like the demanded deadline) is fairly low.

Future Grid systems will consist of tenths of thousands of resource providers operating millions of resources. Having a rising number of resources available, also the probability of finding a migration target increases. Moreover, projects like The Semantic Grid [91] strive on introducing semantically-rich methods in Grid middleware. Systems like BabelPeers [36] allow the semantic description of resources in large scale Grids, implicitly deducting new knowledge from available resource descriptions. Having such semantic methods available, the knowledge about compatibilities does no longer have to be specified entirely by the compatibility profile. Instead existing knowledge in the Grid can be harnessed for finding migration targets.

Bibliography

[1] A. Andrieux et al. Web Services Agreement Specification (WS-Agreement). `http://www.gridforum.org/Meetings/GGF11/Documents/draft-ggf-graap-agreement.pdf`, 2004.

[2] A. Sahai et al. Specifying and Monitoring Guarantees in Commercial Grids through SLA. Technical Report HPL-2002-324, Internet Systems and Storage Laboratory, HP Laboratories Palo Alto, 2002.

[3] A.C. Barbosa, J. Sauvé, W. Cirne, M. Carelli. Independently Auditing Service Level Agreements in the Grid. In 11th HP OpenView University Association Workshop (HPOVUA), 2004.

[4] AssessGrid Project Homepage. `http://www.assessgrid.eu`, 2006.

[5] B. Linnert and L.O. Burchard and F. Heine and M. Hovestadt and O. Kao and A. Keller. The Virtual Resource Manager: An Architecture for SLA-aware Resource Management. In 4th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid), Chicago, IL, USA, 2004.

[6] V. Bohessian, C.C. Fan, Paul S. LeMahieu, M.D. Riedel, Lihao Xu, and J. Bruck. Computing in the RAIN: A Reliable Array of Independent Nodes. IEEE Transactions on Parallel and Distributed Systems, 12(2), 2001.

[7] M. Brune, A. Keller, and A. Reinefeld. Resource Management for High-Performance PC Clusters. In 7th International Conference HPCN Europe, Amsterdam, The Netherlands, volume 1953 of Lecture Notes in Computer Science. Springer Verlag, 1999.

[8] L.O. Burchard, F. Heine, M. Hovestadt, O. Kao, A. Keller, and B. Linnert. A Quality-of-Service Architecture for Future Grid Computing Applications. In 13th International Workshop on Parallel and Distributed Real-Time Systems, Denver, CO, USA, 2005.

[9] C. Dumitrescu, I. Foster. GRUBER: A Grid Resource SLA-based Broker. In EuroPar, Lisboa, Portugal, 2005.

[10] C. Dumitrescu, I. Raicu, I. Foster. DI-GRUBER: a Distributed Approach in Grid Resource Brokering. In Supercomputing, Seattle, WA, USA , 2005.

[11] CCS: Computing Center Software. http://wwwcs.uni-paderborn.de/pc2/index.php?id=19, 2006.

[12] Condor Project. http://www.cs.wisc.edu/condor, 2006.

[13] Corporation for National Research Initiatives (CNRI). Gigabit Testbed Initiative Final Report. http://www1.cnri.reston.va.us/gigafr, 1996.

[14] K. Czajkowski, S. Fitzgerald, I. Foster, and C. Kesselman. Grid Information Services for Distributed Resource Sharing. In 10th IEEE International Symposium on High-Performance Distributed Computing (HPDC-10), 2001.

[15] D. Gannon, K. Chiu, M. Govindaraju, A. Slominski. A Revised Analysis of The Open Grid Services Architecture. Technical report, Department of Computer Science, Indiana University, Bloomington, IN, 2002.

[16] D. Gannon, K. Chiu, M. Govindaraju, A. Slominski. An Analysis of The Open Grid Services Architecture. Technical report, Department of Computer Science, Indiana University, Bloomington, IN, 2002.

[17] D. Jackson, Q. Snell, M. Clement. Core Algorithms of the Maui Scheduler. In 7th Int'l Workshop on Job Scheduling Strategies for Parallel Processing, Cambridge, MA, USA, 2001.

[18] Darwin Homepage at Apple Computers. http://www.opensource.apple.com/darwinsource, 2006.

[19] Distributed Management Task Force (DMTF). http://www.dmtf.org, 2006.

[20] Enterprise Grid Alliance (EGA). http://www.gridalliance.org, 2006.

[21] Enabling Grids for E-SciencE (EGEE) Project. http://public.eu-egee.org, 2006.

[22] I. Foster et al. Software Infrastructure for the I-WAY High Performance Distributed Computing Experiment. In Proc. of the 5th IEEE Symposium on High Performance Distributed Computing, pages 562–571, 1997.

[23] K. Windisch et al. A Comparison of Workload Traces from Two Production Parallel Machines. In 6th Symposium Frontiers Massively Parallel Computing, pages 319–326, 1996.

[24] M. Haji et al. A snap-based community resource broker using a three-phase commit protocol: a performance study. The Computer Journal, 48:333–346, 2005.

[25] The EuroGrid Project. http://www.eurogrid.org, 2006.

[26] European Commission. Commission grants 52 million euro to boost use of GRID. http://europa.eu.int/rapid/pressReleasesAction.do?reference=IP/04/1072, 2006.

[27] F. Heine and M. Hovestadt and O. Kao. HPC4U: Providing Highly Predictable and SLA-aware Clusters for the Next Generation Grid. In 4th Cracow Grid Workshop, Cracow, Poland, 2004.

[28] I. Foster. A Guide to Gara, 2000.

[29] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. International Journal of Supercomputer Applications, 11(2):115–128, 1997.

[30] Foster, I., Kesselman, C., Nick, J. and Tuecke, S. The Physiology of the Grid: Open Grid Services Architecture for Distributed Systems Integration. Technical report, Argonne National Laboratory, University of Chicago, University of Southern California Marina Del Rey, IBM Corp. Poughkeepsie, http://www.globus.org/alliance/publications/papers/ogsa.pdf, 2002.

[31] Global Grid Forum. http://www.ggf.org, 2006.

[32] Globus Alliance: Globus Toolkit. http://www.globus.org, 2006.

[33] Gregory Pfister. In Search of Clusters. Prentice Hall, 1997.

[34] H. Bal et al. Next Generation Grids: European Grids Research 2005-2010. ftp://ftp.cordis.lu/pub/ist/docs/ngg_eg_final.pdf, 2003.

[35] H. Bal et al. 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, 2004.

[36] F. Heine. P2P based RDF Querying and Reasoning for Grid Resource Description and Matching. PhD thesis, University of Paderborn, Germany, 2006.

[37] F. Heine, M. Hovestadt, O. Kao, and A. Keller. Provision of Fault Tolerance with Grid-enabled and SLA-aware Resource Management Systems. In Parallel Computing Conference, Malaga, Spain, 2005.

[38] F. Heine, M. Hovestadt, O. Kao, and A. Keller. SLA-aware Job Migration in Grid Environments. In Lucio Grandinetti, editor, Grid Computing: New Frontiers of High Performance Computing. Elsevier, 2005.

[39] F. Heine, M. Hovestadt, O. Kao, and A. Streit. On the Impact of Reservations from the Grid on Planning-Based Resource Management. In Intl. Workshop on Grid Computing Security and Resource Management (GSRM), Atlanta, GE, USA, 2005.

[40] Felix Heine, Matthias Hovestadt, and Odej Kao. Towards Ontology-Driven P2P Grid Resource Discovery. In 5th International Workshop on Grid Computing, pages 76–83, 2004.

[41] Hirlam (HIgh Resolution Limited Area Model) Programme. `http://hirlam.org`, 2006.

[42] M. Hovestadt. Fault Tolerance Mechanisms for SLA-aware Resource Management. In Workshop on Reliability and Autonomic Management of Parallel and Distributed Systems (RAMPDS-05), at 11th IEEE International Conference on Parallel and Distributed Systems (ICPADS2005), Fukuoka, Japan, 2005.

[43] M. Hovestadt. Operation of an SLA-aware Grid Fabric. Journal of Computer Science, 2(6):550–557, 2006.

[44] I. Foster. What is the Grid? A Three Point Checklist. Technical report, Argonne National Laboratory and University of Chicago, 2002.

[45] I. Foster. A Globus Toolkit Primer. Or, Everything You Wanted to Know about Globus, but Were Afraid To Ask. Technical report, Argonne National Laboratory, 2005.

[46] C. Kesselman I. Foster. The globus project: A status report. Proc. IPPS/SPDP 1998 Heterogeneous Computing Workshop, pages 4–18, 1998.

[47] S. Tuecke I. Foster, C. Kesselman. The nexus approach to integrating multithreading and communication. Journal of Parallel and Distributed Computing, 37:70–82, 1996.

[48] S. Tuecke I. Foster, C. Kesselman. The anatomy of the grid: Enabling scalable virtual organizations. International Journal of Supercomputer Applications, 15(3), 2001.

[49] I. Foster and C. Kesselman (Eds.). The Grid: Blueprint for a New Computing Infrastructure. Morgan Kaufmann, 1998.

[50] I. Foster et al. A Distributed Resource Management Architecture that Supports Advance Reservations and Co-Allocation. In 7th International Workshop on Quality of Service (IWQoS), London, UK, 1999.

[51] IBM Tivoli Workload Scheduler LoadLeveler. `http://www.ibm.com/systems/clusters/software/loadleveler.html`, 2006.

[52] Internet Engineering Task Force (IETF). `http://www.ietf.org`, 2006.

[53] D. Snelling J. Almond. Unicore: uniform access to supercomputing as an element of electronic commerce, future generation computer systems. Future Generation Computer Systems, 15:539–548, 1999.

[54] J. MacLaren et al. 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`, 2003.

[55] J. MacLaren, R. Sakellariou, K. T. Krishnakumar, J. Garibaldi, and D. Ouelhadj. Towards Service Level Agreement Based Scheduling on the Grid. In 14th International Conference on Automated Planning & Scheduling (ICAPS 04), Whistler, British Columbia, Canada, 2004.

[56] J. Padgett, K. Djemame, P. Dew. Grid-based SLA Management. In European Grid Conference (EGC), Amsterdam, The Netherlands, 2005.

[57] J. Padgett, K. Djemame, P. Dew. Grid Service Level Agreements combining resource negotiation and predictive run-time adaptation. In 4th All Hands Meeting, Nottingham, UK, 2005.

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

[59] J.P. Banatre et al. Future for European Grids: GRIDs and Service Oriented Knowledge Utilities. Technical report, Expert Group Report for the European Commission, Brussel, 2006.

[60] K. Czajkowski, A. Dan, J. Rofrano, S. Tuecke, and M. Xu. Agreement-based Grid Service Management (OGSI-Agreement). Technical report, Global Grid Forum, GRAAP-WG, 2003.

[61] K. Czajkowski, D. Ferguson, I. Foster, J. Frey, S. Graham, T. Maguire, D. Snelling, S. Tuecke. From Open Grid Services Infrastructure to WS-Resource Framework: Refactoring & Evolution. Technical report, Fujitsu Ltd., IBM, University of Chicago, 2004.

[62] K. Czajkowski et al. SNAP: A Protocol for Negotiating Service Level Agreements and Coordinating Resource Management in Distributed Systems. In U. Schwiegelshohn (Eds.) D.G. Feitelson, L. Rudolph, editor, Job Scheduling Strategies for Parallel Processing, 8th InternationalWorkshop, Edinburgh,, 2002.

[63] A. Keller and A. Reinefeld. Anatomy of a Resource Management System for HPC Clusters . Annual Review of Scalable Computing, Singapore Univ. Press, 3, 2001.

[64] L. Srinivasan, J. Treadwell. An Overview of Service-oriented Architecture, Web Services and Grid Computing. Technical report, Hewlett-Packard Software Global Business Unit, 2005.

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

[66] B. Linnert, L.O. Burchard, F. Heine, H.U. Heiss, M. Hovestadt, O. Kao, A. Keller, and J. Schneider. The Virtual Resource Manager: Local Autonomy versus QoS Guarantees for Grid Applications. Future Generation Grids, 2005.

[67] M. Hovestadt et al. Scheduling in HPC Resource Management Systems: Queuing vs. Planning. In Job Scheduling Strategies for Parallel Processing: 9th International Workshop, JSSPP, Seattle, WA, USA, 2003.

[68] Maui Scheduler Administrator Guide. `http://www.clusterresources.com/products/maui/docs/mauiadmin.shtml`, 2006.

[69] Microsoft Compute Cluster Server (CCS) Homepage. `http://www.microsoft.com/windowsserver2003/ccs/default.mspx`, 2006.

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

[71] S. Mueller. Etablierung lokaler Sicherheitsmechanismen im Kontext einer Grid-Integration von Resource Management Systemen, diploma thesis, 2006, to appear.

[72] National Coordination Office for Networking and Information Technology Research and Development. http://www.nitrd.gov, 2006.

[73] National Center for Supercomputer Applications. http://www.ncsa.uiuc.edu, 2006.

[74] NextGRID Project Homepage. http://www.nextgrid.org, 2006.

[75] Network Measurements Working Group (NM-WG). https://forge.gridforum.org/sf/projects/nm-wg, 2006.

[76] Organization for the Advancement of Structured Information Standards (OASIS). http://www.oasis-open.org, 2006.

[77] Open Grid Forum (OGF). http://www.ogf.org, 2006.

[78] Object Management Group (OMG). http://www.omg.org, 2006.

[79] Web Services Interoperability Organization (WS-I). http://www.ws-i.org, 2006.

[80] Open Grid Services Architecture Workinggroup of the Global Grid Forum (OGSA-WG). https://forge.gridforum.org/projects/ogsa-wg, 2006.

[81] Open Grid Services Infrastructure Workinggroup of the Global Grid Forum (OGSA-WG). https://forge.gridforum.org/projects/ogsi-wg, 2006.

[82] OpenBSD Homepage. http://www.openbsd.org, 2006.

[83] OpenPBS Homepage. http://www.openpbs.org, 2006.

[84] OurGrid Project Homepage. http://www.ourgrid.org, 2006.

[85] Peter H. Salus. Casting the Net. From ARPANET to Internet and Beyond. Addison-Wesley, 1995.

[86] P.Z. Kunszt. The Open Grid Services Architecture: A Summary and Evaluation. Technical report, CERN, Geneva, 2002.

[87] S. Venugopal R. Buyya, D. Abramson. The Grid Economy. In M. Parashar and C. Lee, editors, Proceedings of the IEEE, volume 93 of Special Issue on Grid Computing, pages 698–714. IEEE Press, New Jersey, USA, 2005.

[88] A. Roy and V. Sander. GARA: A Uniform Quality of Service Architecture. In Jennifer M. Schopf Jarek Nabrzyski and Jan Weglarz, editors, Grid Resource Management: State of the Art and Future Trends, pages 377–394. Kluwer Academic Publishers, 2003.

[89] Resource Specification Language (RSL) v1.0. http://www.globus.org/toolkit/docs/2.4/gram/rsl_spec1.html, 2006.

[90] S. Graham, I. Robinson, et al. Web Services Resource Framework Primer. Technical report, OASIS Open, 2005.

[91] Semantic Grid Project. http://www.semanticgrid.org, 2006.

[92] Sun Grid Engine (SGE) Homepage. http://gridengine.sunsource.net, 2006.

[93] L. Smarr and C.E. Catlett. Metacomputing. Communications of the ACM, 35(6):44–52, 1992.

[94] Sun Solaris. http://www.sun.com/software/solaris, 2006.

[95] The OGSI.NET Project. http://www.cs.virginia.edu/~gsw2c/ogsi.net.html, 2006.

[96] The WSRF.NET Project. http://www.cs.virginia.edu/~gsw2c/wsrf.net.html, 2006.

[97] UNICORE Forum e.V. http://www.unicore.org, 2006.

[98] V. Yarmolenko et al. SLA Based Job Scheduling: A Case Study on Policies for Negotiation with Resources. In 4th All Hands Meeting, Nottingham, UK, 2005.

[99] VOMS Architecture v1.1. Web Page, 2002.

[100] V. A. Vyssotsky and F. J. Corbató. Introduction and overview of the Multics system. In AFIPS Conference Proceedings 27, pages 185–196, 1965.

[101] World Wide Web Consortium (W3C). http://www.w3.org, 2006.

[102] White Rose Grid Homepage. http://www.wrgrid.org.uk, 2006.