Adding Cycle Scavenging Support to the Koala Grid Resource Manager

Bart Grundeken

Delft University of Technology

Adding Cycle Scavenging Support to the Koala Grid Resource Manager

Master's Thesis in Computer Science

Parallel and Distributed Systems Group
Faculty of Electrical Engineering, Mathematics, and Computer Science
Delft University of Technology

Bart Grundeken

February 1, 2009

Author
Bart Grundeken

Title
Adding Cycle Scavenging Support to the Koala Grid Resource Manager

MSc presentation
February 11, 2009

Graduation Committee

prof.dr.ir. H.J. Sips (chair)	Delft University of Technology
ir.dr. D.H.J. Epema	Delft University of Technology
O.O. Sonmez, MSc.	Delft University of Technology
dr. K.V. Hindriks	Delft University of Technology

To my parents, Gerard and Meriam

Abstract

Cycle scavenging (CS) is the process of using otherwise idle computational resources to provide large, aggregate, amounts of computational power. It is the core principle of so-called desktop grids and volunteer computing, which use the idle cycles of desktop computers to do computations. However, resources in a multi-cluster grid likewise may have idle time, and so, multi-cluster grids go through periods of low efficiency. In addition, many practical grid applications are of the Bag-of-Tasks (BoT) type, which are large collections of embarrassingly parallel tasks. These require a vast amount of computational power, which multi-cluster grids can provide. Claiming an entire grid for one application would however not be fair to other grid users. In this thesis, we design a system for multi-cluster grids that detects and uses otherwise idle resources, specifically to execute BoTs. By combining CS with the computational power of the grid, we increase grid efficiency, we are able to provide the resources needed by BoTs, and, at the same time, we can guarantee unobtrusiveness to all grid users. The system we design, KOALA-CS, is an extension of the KOALA grid resource manager. We create a framework for submitting CS jobs through KOALA, and implement several policies for fair sharing among such CS jobs. We then evaluate the performance of the complete system, and demonstrate its capabilities, through a series of experiments on a real grid system, the DAS-3. With these experiments, we show that KOALA-CS does not hinder non-CS grid users, that it ensures fair sharing of resources among CS jobs, and that it is robust and fault tolerant.

Preface

Computers have held my interest since a young age. At Delft University of Technology, I was able to extend that interest into practical knowledge for designing complex systems. Distributed computing caught my eye as one of the most challenging fields, and within that field, Grid computing was something I was unfamiliar with and wanted to know more about. The premise of creating a system within such a complex environment as grids, as well as the original problem domain (applying the system to solve complex games), finally, pulled me in. The result is the thesis that know lays before you. I did the research for it in the context of the Virtual Lab for e-Science project. Parts of Chapters 3 and 4 have been submitted to and accepted by the *Ninth IEEE International Symposium on Cluster Computing and the Grid* [35].

Of course, I could not have completed this project alone. First of all, I would like to thank the Grid group at Delft University of Technology: Ozan Sonmez, for his day-to-day guidance and support; Hashim Mohamed, for developing KOALA and helping me understand and extend it; Dick Epema, for his guidance, especially with writing this report; Alex Iosup and all the other members for their input. In addition, I thank Henk Sips for chairing the graduation committee and Koen Hindriks for participating in it. I would also like to thank my fellow students from the 9th floor laboratory, my fellow interns of the ”KPN group”, and all my friends who took an interest in this project. I extend my special gratitude to my family: my brothers and sister-in-law, but foremost my parents, for their support in too many ways to mention. Last, but not least, I thank God for giving me the ability to learn, to improve, and to be creative.

Bart Grundeken

Delft, The Netherlands

February 1, 2009

Contents

Preface v

1 Introduction 1
1.1 Computational Grids and Cycle Scavenging 2
1.2 Related Work 2
1.3 Contributions 3
1.4 Thesis Outline 4

2 Cycle Scavenging in Multi-Cluster Grids 5
2.1 Background 5
2.1.1 Grids 5
2.1.2 The DAS-3 Multi-cluster grid 9
2.1.3 Cycle scavenging 10
2.2 The Koala Grid Resource Manager 11
2.2.1 Architecture of Koala 11
2.2.2 Koala job flow 12
2.2.3 The Koala scheduler 14
2.2.4 Koala runners 16
2.2.5 The Koala component manager 20
2.3 System and Job Models 23
2.3.1 System model 23
2.3.2 Job model 23
2.4 Cycle Scavenging Applications 24
2.4.1 Bag-of-Tasks applications 24
2.4.2 Dummy application 24
2.4.3 Eternity II 25
2.5 Problem Statement 25

3 The Design and Implementation of Koala-CS 27
3.1 The CS Functionality of the Scheduler 28
3.1.1 Resource discovery 28

3.1.2 Fair allocation of resources . . . 28
3.1.3 Grid-level unobtrusiveness . . . 29
3.2 The Extended Koala Component Manager . . . 30
3.2.1 Local job detection . . . 30
3.2.2 Component submission . . . 31
3.3 The Launcher Mechanism . . . 32
3.3.1 Deployment protocol . . . 32
3.3.2 Scheduling control . . . 33
3.3.3 Task management . . . 33
3.3.4 Communications . . . 34
3.4 The CS Runner and Runners Framework . . . 35
3.4.1 Structure of the CS runners framework . . . 36
3.4.2 Runner components . . . 36
3.4.3 Interaction of runner components . . . 38
3.4.4 Handling grow and shrink messages . . . 39
3.4.5 Launcher management . . . 40
3.5 Koala-CS Job Flow Protocol . . . 41
3.5.1 Job submission . . . 41
3.5.2 Job growth . . . 42
3.5.3 Job shrinking . . . 43
3.5.4 Job completion . . . 44
3.6 Koala-CS Fault Tolerance . . . 44
3.6.1 Error and failure prevention . . . 45
3.6.2 Component failure handling . . . 45
3.6.3 System failure handling . . . 47
3.7 The Policies of Koala-CS . . . 47
3.7.1 CS policies . . . 47
3.7.2 Application-level scheduling policies . . . 48

4 Evaluation of Koala-CS **51**
4.1 Methodology and Metrics . . . 51
4.2 The Efficiency of the Launcher Mechanism . . . 52
4.2.1 Experimental setup . . . 52
4.2.2 Results and discussion . . . 52
4.2.3 Conclusion . . . 53
4.3 The Unobtrusiveness of Koala-CS . . . 54
4.3.1 Local job delay . . . 54
4.3.2 Grid job delay . . . 56
4.3.3 Conclusion . . . 57
4.4 The Effect of CS Fair Sharing Policies . . . 58
4.4.1 Experimental setup . . . 58

4.4.2 Results and Discussion 59
4.4.3 Conclusion 63

5 Conclusions and Future Work **65**

References **68**

Chapter 1

Introduction

Cycle scavenging (CS) is the process of using otherwise idle computational resources to provide large, aggregate, amounts of computational power. It is the core principle of so-called desktop grids, which use the idle cycles of desktop computers to do computations. Desktop computers can indeed provide large amounts of computational power [33]. This power would otherwise be wasted as the computer is active but running the screensaver or doing little more than I/O. By making their desktops part of a desktop grid, the desktop owners can drastically increase the efficient use of their computers.

However, desktop computers are not the only systems that can be idle while still consuming energy. The resources of a multi-cluster grid likewise suffer idle time and thus periods of low efficiency. Inefficiency of multi-cluster grids is even greater, because while users normally switch off their desktops when, for instance, leaving work or going to bed, grid users do not — cannot — switch off the grid. In this thesis we design a CS system for multi-cluster grids to tackle this inefficiency.

The type of applications run on desktop grids is the Bag-of-Tasks (BoT). BoTs are large collections of embarrassingly parallel tasks that may require a vast amount of computational power. Grids can provide this power, yet have to deal with multiple users, and usually assigning all the grid's nodes to the same job is impossible. A CS system avoids this problem by claiming only those nodes which would otherwise be idle. That way, we can apply the grid to BoTs effectively, while at the same time providing all the users with a fair share of the computational power.

The system we design in this thesis, KOALA-CS, is an extension of the KOALA grid resource manager. We create a framework for submitting CS jobs through KOALA, and implement several policies for fair sharing among such CS jobs. We then validate the complete system, and demonstrate its capabilities, through a series of experiments on a real grid system, the DAS-3. With these experiments, we show that the CS extension to KOALA is unobtrusive to non-CS grid users, that it ensures fairness among CS jobs, and that it is robust and fault tolerant.

In the remainder of this chapter, we give a brief overview of background information in Section 1.1. We present related work in Section 1.2. Section 1.3 states the contributions of our work. Finally, we give an outline of this thesis in Section 1.4.

1.1 Computational Grids and Cycle Scavenging

Computational grids are groups of computational resources connected through a wide area network. These resources often have heterogeneous capabilities and belong to different organizations, which may impose further heterogeneity as different organizations have, for instance, different security schemes and policies. The core idea of grids is however that these computational resources can work together, as a single coherent system, without the user having to deal with all the heterogeneity. In this thesis, we categorize computational grids as either multi-cluster grids or desktop grids.

Multi-cluster grids consist of multiple computer clusters. Each of these clusters generally consists of a number of nodes, each with one or more processors, connected through a high-speed network backbone. A single node, the head node, is used as an access point to the cluster, while the other nodes are used for computations only. Job execution on the computational nodes of a cluster is controlled by a resource manager or scheduler. In a multi-cluster grid, these clusters are connected in such a way that it is possible to submit jobs from any one cluster to any other cluster if needed. The grid users can sometimes do this directly but it is often more efficient to submit through the grid resource manager. The DAS-3 is such a multi-cluster grid, and KOALA [30, 31] is a grid resource manager. We use the DAS-3, and KOALA, for our research [35].

Desktop grids consist of collections of desktop computers. However, since each desktop is in fact a single node, jobs can only be remotely run on such a computer if it would otherwise be idle. There are quite a few so-called volunteer computing projects, such as as Seti@Home [8], Compute Against Cancer [1], and the Great Internet Mersenne Prime Search [12]. Desktop grids use CS systems, which detect idle desktops and submit jobs to those desktops. When the desktop's user once again needs the desktop, the CS system must ensure that he can, all but immediately, use all the resources of his desktop again, i.e., the CS system must guarantee *unobtrusiveness*.

1.2 Related Work

There are number of systems that use CS techniques to put idle cycles to good use. Most of these systems facilitate desktop grids or volunteer computing, and some provide CS capabilities to multi-cluster grids.

Amongst the CS systems for volunteer computing, BOINC [15] is perhaps the most famous. BOINC supports such projects as Folding@home [3], Rosetta@home [7], and Seti@home [8]. BOINC provides a set of tools that allow for installing client software remotely on large numbers of desktops, and to subscribe to multiple projects. Desktop owners can then specify how their resources are allocated among different BOINC-based projects. Another desktop grid system is Entropia [17]. Entropia uses its binary sandboxing technology for ensuring security and unobtrusiveness. The architecture of Entropia allows for physical node management, resource scheduling, and job management layers.

OurGrid [18] is an open platform that provides a form of peer-to-peer CS. Different research labs can share their idle computational resources. OurGrid uses a peer-to-peer incentive mechanism called Network of Favors, which makes it in the best interest of each participant to donate idle cycles. This mechanism ensures participation and prevents the peer-to-peer problem of free riding. However, each user is represented by an agent, and this agent competes with other agents to schedule the user's jobs. Therefore, OurGrid does not provide fair-share resource allocation among users.

Condor [27], a well-known grid platform, was initially designed as a desktop grid system, but it has also been extended to operate as a batch scheduler on top of cluster systems and as a grid resource manager on top of Globus-based grids [5, 22]. Condor can be used as a CS system for multi-cluster grids by configuring each node such that it can execute Condor jobs when no jobs submitted by the local scheduler are running. Instead of traditional scheduling, Condor uses the ClassAd mechanism to match jobs to resources based on the job's resource requirements. The Up-Down [32] algorithm ensures fair-share resource allocation based on the past resource usage of each user. Using this algorithm, Condor protects the light users against resource monopolization by heavy users.

KOALA-CS does not rely on any historical information to ensure the fair sharing of resources; instead, it dynamically partitions the idle resources evenly among users, in real time. In addition, KOALA-CS does not require the installations (on head or compute nodes) or modifications such as are required by, for instance, Condor. KOALA-CS seamlessly integrates CS into grid-level scheduling.

1.3 Contributions

We aim to alleviate the inefficiency that exists in multi-cluster grids when nodes are idle. To do this, we must provide these nodes with useful work when they would otherwise do nothing. In addition, we recognize the need for large-scale computing power to run BoTs, and the difficulties to run those BoTs on multi-cluster grids. This, in turn, requires facilities for ensuring fairness to all grid users, i.e., that no user with higher-priority jobs is hindered, and that all BoTs get a fair share of the available power.

The major contributions of this thesis are the following:

1. The design, the implementation, and the deployment of KOALA-CS, an extension to KOALA for cycle scavenging, as well as its experimental evaluation and demonstration of its unobtrusiveness, robustness, and reliability, on the DAS-3 testbed.
2. The design and experimental analysis (on the DAS-3 testbed) of two grid-level CS policies: Grid-Equipartition and Site-Equipartition for fair sharing processor power among CS jobs.
3. The design and testing of a framework for combining application-level scheduling with fairness under CS conditions.

1.4 Thesis Outline

The remainder of this thesis is organized as follows. In Chapter 2 we give a more detailed overview of grids (including the DAS-3), cycle scavenging, and the KOALA grid resource manager. In addition, we provide a system and job model for KOALA-CS, and discuss the applications we test it with. We present the design of KOALA-CS in Chapter 3. We discuss the architecture of the new components for KOALA and the extensions of the existing KOALA components. In that chapter we also introduce the policies for scheduling and fair sharing that we implement. In Chapter 4 we describe the experimental evaluation of KOALA-CS. Finally, in Chapter 5, we present our conclusions and opportunities for future work.

Chapter 2

Cycle Scavenging in Multi-Cluster Grids

CS systems were initially designed for desktop grids, with as a goal harnessing the power of otherwise idle computational resources. In multi-cluster grids, this goal remains unchanged; however, the structure of such grids differs significantly from that of desktop grids. In this chapter, we present the complete problem setting of this thesis by providing the reader with all necessary information for understanding our work.

We discuss the background needed to understand our work in Section 2.1. This background includes a more detailed discussion of grids and CS, as well as a description of our testbed, the DAS-3. In Section 2.2, we elaborate on KOALA, the grid resource management system that we extend in this thesis. In Section 2.3, we provide the system and job models that we employ in this thesis. We present the type of applications (BoTs) suitable for a (multi-cluster) CS system, and the specific applications we use for testing in Section 2.4. Finally, we give a detailed problem statement in Section 2.5.

2.1 Background

In this section, we discuss background information needed to understand the rest of this thesis. In Section 2.1.1 we discuss grids in general, while in Section 2.1.2 we discuss DAS-3, which is our test-bed and a real multi-cluster grid. Finally, we cover cycle scavenging in Section 2.1.3.

2.1.1 Grids

There are many definitions of computational grids. The core idea of grids is that they are a collection of interconnected, heterogeneous, geographically and admin-

istratively distributed resources. Ideally, all of these resources should be able to have their own security protocols and other policies, while the grid as a whole should be transparently accessible to all of its users. In practice, most grids are not as heterogeneous as was originally expected, however, grids are not as transparent as was originally hoped either. We give an example of a ”real” grid in Section 2.1.2, the DAS-3 which is the grid we us as a development platform and testbed. In the remainder of his section, we go deeper into three grid issues which are of concern to the work in this thesis: grid heterogeneity, grid organization, and resource availability.

Grid heterogeneity

Grid heterogeneity comes in different forms. First, grids may contain resources with varying capabilities. The ”ideal” grid may consist of a collection of desktops, multi-computer clusters, and specific equipment, among other resources. In reality, grids are often more homogeneous concerning resource capabilities. Multi-cluster grids, which we are concerned with, consist of collections of multi-computer clusters. These clusters may still differ in processor speed, internal memory, secondary memory, number of processor cores, and so forth, which may cause grossly varying performance between clusters.

Second, there can be heterogeneity in the access control on each resource. While some resources may be directly available to schedule jobs on, more often than not jobs are scheduled on resources through a local scheduler of some sort. Most multi-clusters need a local scheduler to handle jobs submitted locally, or there could be contention and other problems on the cluster. How to manage jobs differs from scheduler to scheduler, creating not a few problems for interoperability. Closely tied to this issue is that of security: there may be different schemes in place on different resources. Where one may require a username and password, others may need public and private key-pairs. Also, the credentials can differ from resource to resource.

Third and finally, there is administrative heterogeneity, which, to no small extent, influences the other two issues. Different organizations may control different parts of the grid. This reflects itself in naming conventions, the type and amount of hardware used, utilization limits and other such topics.

All of these forms of heterogeneity have to be dealt with for effective grid operations. Most can be dealt with using grid middleware, such as Globus [5] and Fura [4]. However, a great deal of complexity is eliminated when first and foremost the administrative differences are addressed. A group of resource owners wishing to build a grid may come to clear agreements beforehand, eliminating much of the access control heterogeneity, synchronizing security practices, and agree on the type of resources offered. Such consensus is readily apparent from, for instance, the DAS-3.

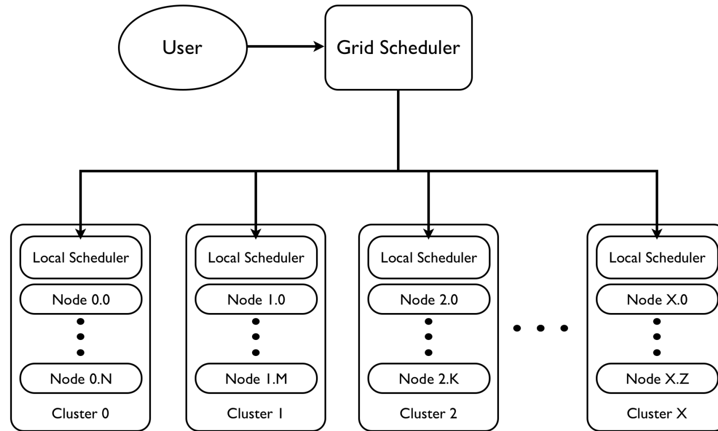

Figure 2.1: A centralized grid architecture

Because of the agreements made between various owners of the clusters that make up the DAS-3, KOALA-CS has only to deal with resource heterogeneity. We discuss the specifications of the DAS-3, or at least those where it is heterogeneous, in Section 2.1.2.

Grid organization

A second point that is important in grids is organization. To deal with the level of complexity and heterogeneity that appears in the grid, its organization needs to be well defined.

One issue regarding access control is the use of local schedulers. Without them, grid schedulers (which can schedule tasks over multiple resources in the grid), have direct control over a resource. Most grids however, have local schedulers in place on every resource. With such grids, any higher-level scheduling device will have to schedule through them. Furthermore, users may often submit jobs directly to the local schedulers. In the coming discussion, we consider the local schedulers as part of the resource they control. ”Direct” submission to the resources is thus equal to submitting to the local scheduler, should the resource have one. We distinguish three grid scheduling schemes: centralized, peer-to-peer, and hierarchical.

A centralized scheduling architecture is depicted in Figure 2.1. Here, we see that a single grid scheduler (also known as a super- or meta-scheduler), has a direct connection to every resource. The users ideally only submit jobs through the grid scheduler. This scheme has all the advantages and disadvantages of other centralized systems: a high measure of control, yet poor scalability. Still, it is commonly used, especially in relatively small grids. KOALA, which we discuss in Section 2.2 is a grid scheduler for such a grid.

The peer-to-peer architecture (see Figure 2.2) is not often implemented explic-

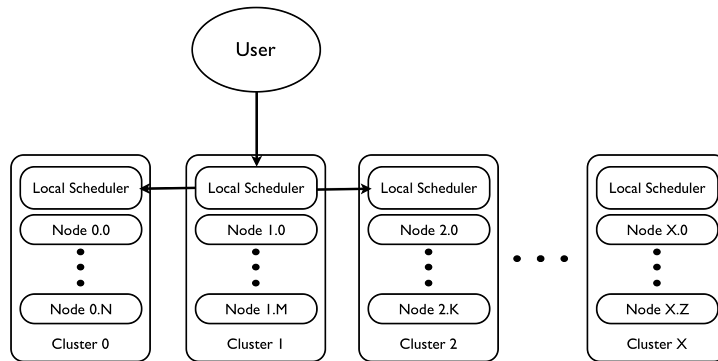

Figure 2.2: A peer-to-peer (grid) architecture

itly, but is often implicitly available in grids. A peer-to-peer grid would have users submit to the local scheduler, which would then try to place the job locally, or move it to a peer resource if there is no room for the job. In reality, this can often be done by the user himself. If he cannot schedule his job on the local scheduler, he can try again at a remote site in the grid. This kind of architecture may also be used for communication between grid schedulers themselves, but then we move into the realm of hierarchical grids.

Hierarchical grids, as illustrated with Figure 2.3, consist of layers of scheduling. Such a grid can be divided into sub-grids, each with their own scheduling practices (Figure 2.3 shows a typical two-level hierarchy with centralized scheduling on both levels). Two or more of these sub-grids can then be connected with some other or the same form of scheduling, forming a sub-grid which can then be connected to others, and so forth. An example of this is Condor Flocking [20], which actually combines lower-level centralized scheduling with a peer-to-peer architecture among sub-grids.

The type of grid architecture that we concern ourselves with is the centralized type. However, the nature of the DAS-3, as we discuss in Section 2.1.2, allows for direct submission to the individual clusters as well. Technically, this allows for ”peer to peer” scheduling by the users themselves, where users can submit their jobs directly to any of the local schedulers. In practice, this means that job submitted through the grid scheduler have to contend with jobs submitted through the local scheduler directly.

Resource availability

In previous sections, we discussed resource and administrative heterogeneity. These have an impact on resource availability, because both influence whether or not a resource is available for use, and if it is the right resource to use. When a resource breaks down, or is otherwise unavailable, it might not always be easily apparent be-

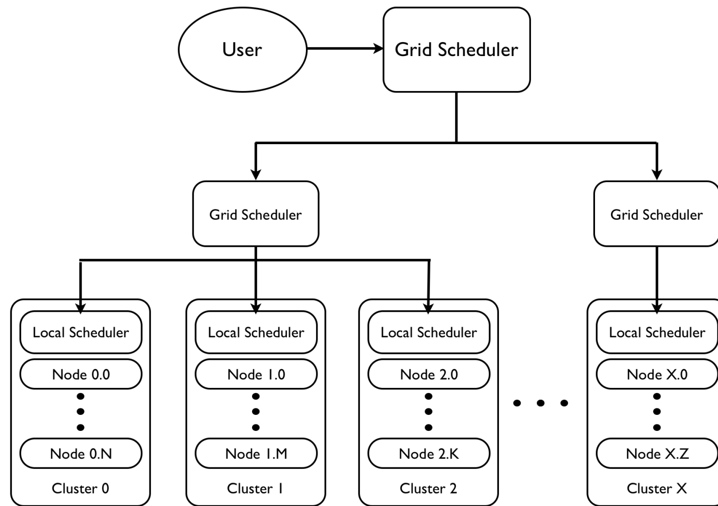

Figure 2.3: A hierarchical grid architecture

cause of administrative boundaries. Also, the resource owner may not bring it back at all. Furthermore, because resources can differ substantially in capabilities, not all resources may be suited for a specific job. In the DAS-3, it is easily detectable if a resource is unusable at the moment, and, except for a few extreme cases, the resources can more or less all be applied to the same kind of problems.

However, resource availability is still dynamic, and this has to be dealt with. If a resource drops out while being in use, there are various ways to proceed. For instance, the job may be checkpointed [25], a complex process that calls for exactly recording the state of a job, moving it to another resource and resuming it from the recorded state. Another approach is to just restart the job (or part of it) elsewhere, which might be inefficient. Finally, the job can just be considered to have failed, which is of course not desirable.

In the remainder of this thesis, it will become clear that the dynamic nature of resource availability is even more apparent in our environment. This has to do with the nature of CS, which we discuss in more detail in Section 2.1.3. We first give a more concrete picture of a real grid, which we already mentioned in this section, namely, the DAS-3.

2.1.2 The DAS-3 Multi-cluster grid

The Distributed ASCI Supercomputer 3 is a five-cluster wide-area distributed system in the Netherlands, designed for doing experimental research on parallel and distributed programming [9]. The five clusters are located at the Delft University of Technology (TUD), Leiden University (LU), Vrije Universiteit, Amsterdam (VU), University of Amsterdam (UvA), and The MultimediaN Consortium (UvA-

Cluster	Nodes	Processor	Storage	Node HDD
VU	85	dual-core, 2.4 GHz	10 TB	250 GB
LU	32	single-core, 2.6 GHz	10 TB	400 GB
UvA	41	dual-core, 2.2 GHz	5 TB	400 GB
TUD	68	single-core, 2.4 GHz	5 TB	250 GB
UvA-MN	46	single-core, 2.4 GHz	3 TB	1.5 TB

Table 2.1: DAS-3 resource capabilities

MN, also located in Amsterdam). We present the capabilities of each cluster in Table 2.1. These capabilities are quite heterogeneous. Although all nodes in the DAS-3 are dual-processor nodes with 4 GB of memory, the processors themselves differ, as do cluster- and node-specific storage. For the inter-node communication there are 1 Gbps and 10 Gbps Ethernet connections in all the clusters. All but the TUD cluster also have high speed Myri-10G connections.

The local schedulers on the individual clusters run the Sun Grid Engine [14] as local resource manager. The WAN connections between the clusters are 1 Gbps connections over Internet provided by the local university. Through these connections, the clusters connect to SURFnet, which provides connections to all other clusters in the DAS-3. However, the StarPlane Project [13] allows the DAS-3 to connect through dedicated 10 Gbps lightpath connections over SURFnet.

2.1.3 Cycle scavenging

As we state in Chapter 1, CS is the process of using otherwise idle computational cycles for useful operations. We mention several CS systems in Section 1.2, such as Condor, BOINC and Entropia. These systems are able to provide large amounts of computational power to their users. Although they differ in several ways, they share the same set of basic principles. We discuss those basics in this section.

Generally, CS systems work with a central server that hands out tasks to client software on idle desktops. The owners of these desktops install the software and register with the appropriate project. The function of the client software is to report when the desktop becomes idle, run the tasks, preempt tasks, and provide a sandbox to the tasks. This sandbox is a safe environment for the task to run in, preventing it from making unwanted changes to the desktop it is running on. Likewise, it shields the task from unwanted interference by the desktop user, who may, intentionally or not, try to affect the task.

CS jobs always have a lower priority than the jobs initiated by the desktop user. Once the user needs resources of his desktop that are currently in use by tasks of the CS job, these tasks must be preempted in such a way that the user is not hindered by it. This property of the CS environment makes these grids even more dynamic than normal grids. Resource availability cannot be guaranteed. This has to be taken into

account when selecting CS applications. Parallel jobs that need multiple resources at the same time and require individual tasks to synchronize, cannot efficiently run in such an environment. Communication between tasks is, in general, difficult because a task may be preempted before it can reply. Also, because tasks may be restarted at another location, their network address may change between communications, making the task that was restarted suddenly unreachable by other tasks. Another property CS applications cannot have is a fixed ordering: no guarantees can be given in a CS grid that one task is finished before the other. Taking all of these issues into account, CS jobs must consist of tasks that do not communicate, have no ordering constraints, and can be preempted without too much work.

Logically, these constraints also apply in multi-cluster grids. However, some can be relaxed. Sandboxing for instance, is not that vital in a system that was meant to be used for remotely issued job execution. Since every job generally incurs some overhead before being able to run (it needs to be scheduled by a local scheduler) and generally has a long, non-interactive, runtime, the time needed to preempt CS jobs can be in the order of a few tens of seconds. Such a delay would be amortized by the job runtime and normal scheduling overhead. Yet the dynamics of resource allocation cannot be underestimated. These primarily influence the CS jobs. Since they are at the bottom of the priority scale, they will be often preempted. Although we do not claim this problem has more or less impact in multi-cluster grids than in desktop grids, it is one of the core issues we have to, and do, deal with in this thesis.

2.2 The Koala Grid Resource Manager

KOALA is a grid resource manager based upon a centralized scheduler structure. It is actively developed at the Delft University of Technology [30, 31]. Originally, it was intended to work with data-heavy grid applications. Its focus was therefore on *co-allocation*, i.e., allocating processing power in multiple clusters simultaneously. However, it evolved to become highly extensible for all kinds of grid applications.

The system we develop, KOALA-CS, is an extension to KOALA. Therefore, we give an overview of KOALA's main features in this section. In addition, we go into more detail regarding those features which we extend with KOALA-CS. We first give an overview of the layered structure and components of KOALA in Section 2.2.1. We then present the basic KOALA job flow in Section 2.2.2. Finally, we discuss each of KOALA's components in Sections 2.2.3 through 2.2.5.

2.2.1 Architecture of Koala

KOALA's architecture is by its very nature highly extensible. Although it works with a centralized scheduler, many aspects of job submission and scheduling are

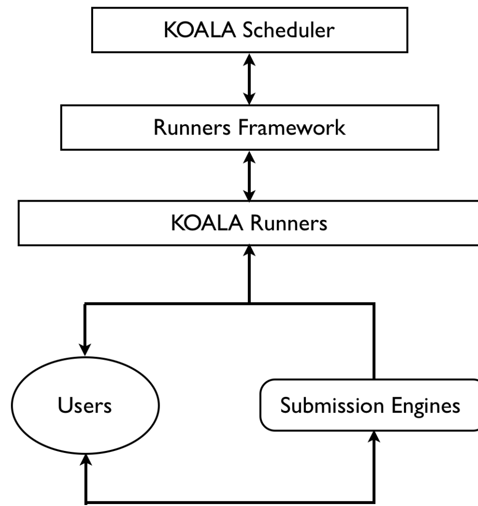

Figure 2.4: The layered structure of KOALA

delegated to other components in the system. These components are organized in four layers, as illustrated in Figure 2.4.

The KOALA *scheduler* is the central point of control in KOALA. It is responsible for detecting and reserving nodes for jobs. However, the actual submission of jobs to these nodes is done by the *runners*. Each runner can be specialized to a specific application (type of job), and it therefore acts as the primary controller for a job. To communicate with the scheduler and interact with the grid, KOALA provides a Runners Framework (RF). KOALA runners are built upon this framework. Finally, we have the *submission engines*, which are third-party tools for job submissions to KOALA. These tools use runners to do the actual submission, and include workload generators, such as GrenchMark [24], and submission scripts, among others. It is beyond the scope of this thesis to discuss all the different types of submission engines, so we do not specifically elaborate on submission engines further. However, we do present more detailed descriptions of the runners and the runners framework, as well as the scheduler, in later sections. An additional component of KOALA, which is not strictly part of any of its layers, is the KOALA Cluster Manager (KCM). The KCM is used by some runners as a tool for submitting to the local schedulers. The KCM is a key component of KOALA-CS, and we therefore discuss this part of KOALA as well.

2.2.2 Koala job flow

In this section we give an overview of the basic job flow protocol in KOALA. KOALA allows jobs to be split up into *components*. Each of these components is typically assigned to a different cluster, and can sometimes be placed independently of other components. This of course allows for more flexibility when

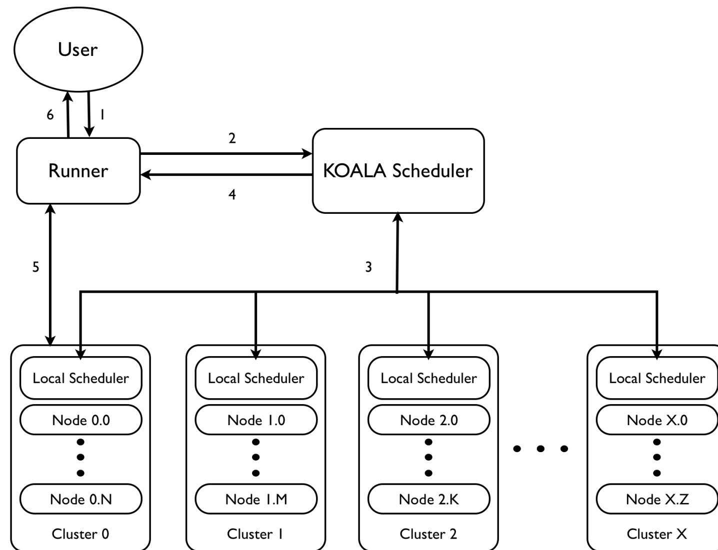

Figure 2.5: KOALA Running a single-component job

scheduling jobs.

The most simple of jobs consists of a single component, to be placed atomically. We illustrate the scheduling of such a job in Figure 2.5. What follows is a high-level description of what happens at every step, where the numbers correspond to those in the figure.

1. The user submits his job to a runner.

2. The runner informs the scheduler of a new job, detailing the job characteristics.

3. The scheduler looks for a suitable site to place the job. Note that the job may remain in the scheduler's queue for a while or may even be rejected if it cannot be placed at all.

4. Once a site is found, the scheduler informs the runner that it may go ahead and submit the job to this site.

5. The runner submits the job to the indicated site (in our case Cluster 0), and waits for it to complete.

6. Upon job completion, the runner gathers the results and presents them to the user.

For all types of jobs, the basic protocol is the same. The runners framework standardizes the message exchange of steps 2 and 4. However, what happens at

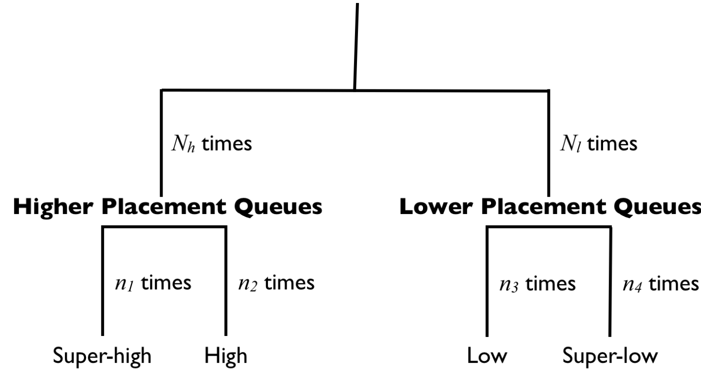

Figure 2.6: The WRR scanning of placement queues

steps 1, 5, and 6 is largely application-specific. Different runners may use different techniques for submission to the clusters. Also, the way users submit jobs to the runners is also not fixed. For other types of jobs, for instance, those that are split over several clusters (and thus consist of several components), the runner may receive multiple site allocations from the scheduler at once, or some components may be placed before others. Handling those situations is the responsibility of the runner (see Section 2.2.4 for more details).

2.2.3 The Koala scheduler

The KOALA scheduler allocates nodes to jobs. It is the central authority in the grid, and is aware of all the clusters in the grid. In this section, we first discuss job placement and node claiming, two phases that a job goes through when it is submitted to the scheduler. These two phases roughly cover steps 2 through 4 in Figure 2.5. Second, we discuss the communication protocol used by other components to communicate with the scheduler.

Job placement

The scheduler has four *placement queues*, which are labelled by priority: *super-high*, *high*, *low*, and *super-low*. A job's priority, as indicated at job submission time, determines in which queue it originally resides. The KOALA scheduler scans the queues at regular intervals using a Weighted Round Robin (WRR) scheme. We illustrate WRR in Figure 2.6.

The super-high and high placement queues are grouped as the higher placement queues, and the low and super-low as the lower placement queues. In each round, the high queues are scanned N_h times, and the low queues N_l times, where $N_h \geq N_l \geq 1$. Each scanning of the high placement queues involves n_1 times scanning the super-high queue, and n_2 times scanning the high queue, where again $n_1 \geq$

$n_2 \geq 1$. Similar for the scanning of the low queues, where the scheduler scans the low queue n_3 times, and the super-low queue n_4 times, with $n_3 \geq n_4 \geq 1$. So, this results in, for instance, the super-high queue being scanned $N_h \times N_1$ times before the low or super-low queues are scanned.

Each time the scheduler scans a job, the scheduler tries to place that job. This means that it tries to find the best cluster(s) to run the job's component(s), according to the job's placement policy. Two of the policies possible in KOALA are Worst Fit (WF), that places components in decreasing size order on clusters with the largest number of idle processors, and Close-to-Files (CF), that places components such that the file transfer time between execution and file sites is minimized. If the scheduler fails, it moves on to the next job. If it succeeds, it moves the job to the *claiming queue*.

The scheduler keeps track of the number of *placement tries* per job. A placement try is an attempt to place the job at a cluster. KOALA has two parameters related to placement tries, which both can be set to ∞. First, it has a maximum number for placement tries. If a job's number of placement tries exceeds this maximum, KOALA aborts the job. Second, KOALA has a value P, which indicates how many placement tries a job can have before the scheduler upgrades the job to a higher placement queue. Note here that the scheduler never upgrades jobs in the super-low queue.

Node allocation and reservation

When a job enters the claiming queue, KOALA estimates its File Transfer Time (FTT) and Job Start Time (JST). The scheduler guarantees that the needed nodes are available at JST. For data-heavy jobs, this is a challenge, since FTT may be large and claiming processors while file transfer is in progress, but the job is not running, is wasteful. If possible, the scheduler therefore reserves the needed processors at the cluster where the job is placed. This requires the local scheduler of that cluster to have such functionality, which is often not the case. Therefore, KOALA defines the Incremental Claiming Policy (ICP). Globally, this policy tries to claim the nodes for job at the cluster where it was placed at Job Claiming Time (JCT), some point in time before JST. If this is impossible, it tries to place it somewhere else using the job's placement policy, and claim the nodes there at a JCT closer to the JST. If JCT and JST come to close, node claiming fails and the job in question returns to the placement queue it came from. Note that the actual submission of the components to the clusters where they are placed, is the responsibility of the runner. Once the nodes are claimed, the scheduler informs the runner, and the runner submits the correct component to those nodes.

command#job-identifier#message body

Figure 2.7: The KOALA message format

Inbound communications

To communicate with the scheduler, the other KOALA components use a standardized message protocol. A message to the KOALA scheduler is formated as in Figure 2.7. The first field, *command*, is one of the commands from Table 2.2. The *job-identifier* consists of the job ID, component ID, and job try count, separated by colons. This is used, obviously, to identify the job and component the message concerns, but also which job try it belongs to. If the job try count in the message does not reflect the latest job try, the message is discarded. The contents of the field *message body* differs for each message type.

In Table 2.2, we see the commands that can be used in messages to the scheduler. The `JOB_NEW` command comes with a variety of job information, packed with the message to inform the scheduler of the necessary information for running the job. This information includes such things as the placement policy, whether or not to use co-allocation for the job, user name, clusters to exclude for this job, intended job priority, job type, and port number of the runner. In addition, it must contain a job description in the job description language RSL [11]. Job type requires some additional explanation. With job type, the runner can indicate how the scheduler should treat the job. A job can be of *rigid* type, where it is allocated nodes and once it has taken those nodes, it has to complete the job with those nodes. Another type is *malleable*, where the job may be allocated new nodes, as well as be mandated to release nodes, during runtime. With the `JOB_NEW` command, the job-identifier is irrelevant, since the scheduler replies by sending a `JOB_ID` message to the runner, informing the runner of its job's ID (see Table 2.3).

`JOB_ABORT`, `JOB_DONE`, and `JOB_RESTART` merely inform the scheduler of the fact that the runner is aborting, respectively done with or restarting, its job. With a restart, the scheduler increases the job try count that is sent with every message, since this indicates with which run a message is associated. When a job is done or aborted, the scheduler removes it from its queues.

Finally, `ACK_GROW` and `CLEAR_COMPONENT_RESERVATION` are used to clear KOALA's node reservations for the job in question. The latter is the only command where the component ID from the job identifier is of any relevance to the scheduler, since node reservations are made per job component. Note that these messages are only sent as replies to messages from the scheduler to the runners (see Table 2.3).

2.2.4 Koala runners

KOALA runners are based on the KOALA Runners Framework. The RF is a library of Java classes that allows a runner to communicate with the scheduler. The RF

Command	Description
JOB_NEW	Registers a new job with the scheduler.
JOB_ABORT	Indicates a job is aborted. Removes it from the scheduler's queues.
JOB_RESTART	Indicates a job is restarted.
JOB_DONE	Indicates job completion.
ACK_GROW	Acknowledges receiving one or more JOB_GROW messages. Clears the associated reservations.
CLEAR_COMPONENT_RESERVATION	Clears the reservations for the indicated component.

Table 2.2: Scheduler commands

provides an abstract class, AbstractRunner that contains functionality all runners share, and every implementation of a specific runner must extend this class. This class is used by another RF class, the RunnerListener, that acts as a message server. The RL, combined with the a subclass of AbstractRunner, makes up a complete KOALA runner. In addition to handling communications between the KOALA components, the RF provides access to KOALA's data management facilities, its Job Description Language (JDL) parsing, and various other utilities.

Runner operations

The user can start a runner with a set of command-line parameters that vary from runner to runner, but generally includes a Job Description File (JDF) in a JDL that the runner can parse. This JDF describes the job parameters. The runner then starts acquiring nodes through the scheduler, and executing the job. Upon job completion, it is responsible for clean up, gathering, and presenting job results in some form or the other. Once active, a runner acts as a message server. Most runners are purely event-driven, acting only in reaction to these messages.

Messages to the runner must be formatted in the same way as those to the scheduler. Therefore, they likewise follow Figure 2.7, and again, if the job try count does not match what the runner knows about the job, the message is discarded. The list of commands that can go in messages to the runner differ from those for the scheduler. Table 2.3 shows the commands that the runner can receive.

The AbstractRunner class provides basic responses to most of these commands. For some commands, there is no difference from runner to runner. With JOB_ID, it assigns this ID to the internal representation it has of the job. The response to COMPONENT_PLACED is to assign the indicated cluster and number

Command	Description
JOB_ID	Indicates that the message contains the job ID the scheduler has assigned to the runner's job.
JOB_RESTART	Tells the runner to restart the job (and thus increase its try count by one).
JOB_ABORT	Tells the runner to immediately abort the job.
COMPONENT_PLACED	Indicates at which location a component of the runner's job is placed. The body contains the name of the cluster and the number of nodes reserved there.
SUBMIT_COMPONENT	Tells the runner to submit a previously placed component.
TRANSFER_FILES	Tells the runner to start staging in relevant files.
JOB_SHRINK	Tells the runner to decrease its node allocation at a certain cluster for a certain number, both of which contained in the message body.
JOB_GROW	Informs the runner that it can claim an additional number of nodes at a certain cluster. Again, the message body contains the name of the cluster and the number of nodes.
APPLICATION_CALLBACK	Indicates that this is a message from the application and not from the scheduler. The body contains the actual message from the runner's job.
REPORTER	Indicates that this is a status message from the scheduler.

Table 2.3: Runner commands

of nodes to the runner's representation of the component. It then replies with a CLEAR_COMPONENT_RESERVATION message (see Table 2.2). With REPORTER, the default response is to print the message. It triggers no further action.

Other commands consist of a response that is partially fixed, partially application-dependent. These include JOB_ABORT and JOB_RESTART. Both trigger an immediate termination of the job. However, JOB_RESTART also increases the job try count by one and sets the job up for a new run. How the actual termination is accomplished is left to the runner programmer. JOB_GROW is only of interest to runners that deal with malleable jobs. Whatever the response is (that will generally include some form of claiming new nodes), it must include sending an ACK_GROW command (see Table 2.2).

Finally, the SUBMIT_COMPONENT, TRANSFER_FILES, JOB_SHRINK, and APPLICATION_CALLBACK commands are completely dependent on the application; however, they must naturally trigger a logical response to the information in the message. A JOB_SHRINK message, for instance, should not trigger the opposite response of actually increasing the runner's node allocation.

Application-specific operations

A KOALA runner consists, as mentioned before, of a AbstractRunner subclass wrapped in a RunnerListener object. The AbstractRunner subclass must provide application-specific mechanisms. Some messages to the runner must be dealt with on an application-specific basis.

AbstractRunner provides several abstract methods that its sub-classes should override in order to handle these messages. While an overview of these methods would be superfluous here, we do give a list the key issues that a runner has to deal with.

Placement Procedure This varies from application to application. Some require all job components to be placed simultaneously or not at all, others can start computing when only one component can be placed.

Deploying Order The order in which components are placed also differs between applications. Some need synchronous placement of components, with others the order is irrelevant.

Application-Level Scheduling A significant number of applications benefit from application specific scheduling. Here, instead of sending job components to the scheduler and letting the scheduler place them, the runner can ask the scheduler for the execution sites only, then map the components to these sites using some application-level policy.

Wide Area Communication between Components For jobs where the components need to communicate, such as parallel applications, the runner needs

to link up with the appropriate communication libraries. Again, there are applications where components need not to communicate, so those runners do not need to implement these communication facilities.

Fault Tolerance The runner needs to deal with execution site failures. It must provide for appropriate responses to sudden job termination and similar events. Errors generated by the application itself are passed by the RL to the subclass of `AbstractRunner`, and it needs to deal with these errors, otherwise, the default framework action is executed, resulting in runner and job termination.

2.2.5 The Koala component manager

The KOALA Component Manager (KCM) can be used by runners for submitting job components to the local schedulers (of course, only after the KOALA scheduler has placed the job on the associated cluster). It is a relatively new part of KOALA, and is currently being developed as a part of KOALA version 2. It uses the DRMAA [2] interface of the local resource manager, which is standardized, to acquire nodes for the runner. The runner can then decide what to do with these nodes, and how to schedule the tasks the component exists of over these nodes. The KCM therefore allows the runner to bypass the local scheduler, granting the runner more flexibility with scheduling.

Component submission using a KCM

As the name suggest, the KCM handles the components of a KOALA job. For each component, there is one KCM. When a runner submits a component to a cluster (i.e., step 5 of Figure 2.5) , it places a KCM on the head node of that cluster. We illustrate the process in Figure 2.8, as it goes through the following steps.

1. The runner places a KCM on the head node of the cluster where the component is to be placed. The runner provides this KCM with the runner's port number and host name, as well as the number of nodes to claim, and for how long they should be claimed (i.e., the wallclock-time of the component).

2. The KCM submits one or more placeholder scripts to the local scheduler, using that scheduler's DRMAA interface.

3. The local scheduler schedules the placeholder script(s) to a set of its nodes.

4. The placeholder script reports the hostname of its node to the KCM.

5. The KCM compiles a list of the node hostnames it receives from the placeholder scripts and sends them to the runner.

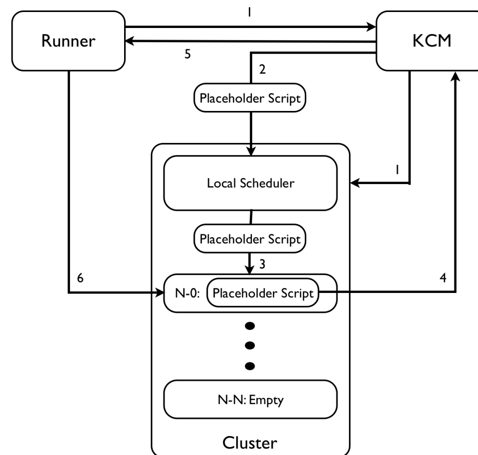

Figure 2.8: Component submission with a KCM

6. The runner uses the node hostnames to submit jobs directly to the nodes in question.

The advantages of this approach include, among others, an increased amount of control over job submission. Once the nodes are known to the runner, it may then schedule the individual tasks of its job to those nodes as the runner sees fit. It may actually schedule more than one task to the same node, while otherwise, all these tasks would have to be submitted separately to the local scheduler, incurring scheduling overhead for each of them. Using a KCM therefore improves both job efficiency and runner control over the execution.

KCM Communications

As stated, a KCM offers more control over job execution. One factor of that control is that during runtime, the allocation of nodes can be actively reduced and increased. In Table 2.4, we list the commands that the runner can send to the KCM to do so. These commands need to be packed in a message with a simple, fixed format. The message must contain first a command and then the body, separated by a colon.

Most runners will not use the `101` command, since they run fixed jobs. Command `100`, however, is commonly used to shut the KCM down when it is no longer needed. It should be noted that a KCM simply schedules the placeholders to the nodes, regardless of cluster capacity. Therefore, runners that *do* use the `101` command, may need to check for availability of nodes.

The KCM also sends messages to the runner. These messages follow a format of first the command and then the body, separated by a percentage sign. The command

Command	Desciption
`100`	Tells the KCM to reduce the node allocation by the number included in the message body. In effect, it causes the KCM to terminate that number of placeholder scripts, thus, as far as the local scheduler is concerned, releasing that node. Note that the KCM terminates only the placeholder scripts; the runner must terminate any tasks of its job running on that node. If this command reduces the node allocation to zero, the KCM terminates itself.
`101`	Tells the KCM to increase its node allocation by the number indicated in the body of the message. The KCM responds by scheduling that number of placeholder scripts to the local scheduler through DRMAA.

Table 2.4: Message interface to the KCM

Command	Description
`STATUS`	Indicates that the body of the message contains the KCM's status: still active or just finished.
`PORT`	Informs the runner of the KCM's port number for incoming communications.
`NODES`	Indicates that the body contains a comma-seperated list of the nodes currently allocated to the KCM.
`ERROR`	Indicates that the KCM terminated with an error, a description of which is included in the message body.

Table 2.5: Messages from the KCM to the Runner

included in such a message must be one from Table 2.5. However, the KCM wraps this message in a default KOALA messages (formatted as in Figure 2.7), with as command the `APPLICATION_CALLBACK` (see Table 2.3).

The `NODES` message is the most interesting to the runner. It indicates which nodes are currently claimed by the KCM for the runner with placeholder scripts. The runner can schedule its job over these nodes. `PORT` is primarily of importance to runners which might increase or decrease their node allocations at runtime, which they do through the commands in Table 2.4; however, all runners need this port if they need to terminate the KCM before the KCM's wall-clock time is over. Note that since the runner deploys the KCM, it is aware of its hostname. The other messages only inform the runner of the KCM activities, and will generally not generate a runner response, except perhaps for redeploying the KCM if needed.

2.3 System and Job Models

In this section, we present the model of a CS grid that we use in this thesis. Some of the terminology contained in this section we used in previous sections already, however, here we formalize them to facilitate further discussion about the system. Section 2.3.1 contains are system model and definitions, while Section 2.3.2 presents the job model.

2.3.1 System model

Our system model is that of a centralized grid (see Figure 2.1). The grid exists of a number of *cluster*s or *host*s, which each consist of a number of *node*s or *resource*s. One of these nodes functions as a head node, and every cluster has a *local resource manager* or *local scheduler*. Users can submit jobs directly to these local schedulers, which will then schedule the job over nodes belonging to that cluster. We call such jobs *local jobs*. Should a user wish to use potentially the entire grid for his jobs, he needs to submit through the centralized *grid scheduler* or *grid resource manager*. Analogous to the term ”local job”, we call any job submitted through the grid scheduler a *grid job*. Whether submitting a grid job or a local jobs, the user submits the job from one of the head nodes in the grid. Likewise, we assume the user has the needed security credentials to access the nodes on all the clusters he wishes to use.

2.3.2 Job model

A *job* is a single run of an application as defined by the user. Each job consists of a number of *task*s, where each task is a part of the job that can be run independently of other parts of the job. All jobs are considered to be *malleable*. With malleable jobs, the number of resources allocated may vary during the runtime of the job. This means the system will have to respond to the allocation of more resources to the job (*job growth*), as well as the forced release of resources (*job shrinking*). The allocation of resources to a malleable job may be reduced if those resources are needed by other jobs of higher priority. Likewise, the number of nodes may increase as the number of idle nodes in the system increases. CS jobs have the added property that they may, at times, have no resources allocated to them (CS jobs can be ”shrunk” to size zero) at all, without being altogether cancelled. Also, CS jobs can in theory be allocated a number of nodes up to the limits of the grid.

In relation to the two types of jobs named in Section 2.3.1 (local and grid jobs), our CS jobs are always of the grid variety. Thus, they are always submitted through the grid scheduler. However, CS jobs always have the lowest possibly priority, meaning that they have to release nodes as soon as any other, non-CS, job requires them. To make an easy distinction between the two, we refer to CS grid jobs simply

as *CS jobs* and refine the term "grid job" as meaning any *non-CS* job submitted through the grid scheduler.

2.4 Cycle Scavenging Applications

In this section we discuss the applications we run on KOALA-CS. In Section 2.4.1 we discuss the general type of applications. In the remaining subsections we discuss our two testing applications: the dummy application in Section 2.4.2, and the Eternity II puzzle solver in Section 2.4.3.

2.4.1 Bag-of-Tasks applications

We limit the applications for KOALA-CS to those of the Bag-of-Task (BoT) variety. BoTs basically consist of a (very large) set of tasks that are each fully independent of the others. In other words, these are embarrassingly parallel applications, and thus ideally suited for CS grids (see Section 2.1.3). Iosup et al. [23] identified BoTs as the predominant application type in industrial grids today. Applications include medical research such as molecular docking [36], examining large data sets, and basically any application which requires searching through a large set of parameters.

A common subtype of the BoT is the Parameter Sweep Application (PSA). The difference between tasks in PSAs lies only in their input parameters. Every task uses the same executable files. The applications we test on KOALA-CS fit into this category.

2.4.2 Dummy application

For several tests, we use a dummy application. This application waits for a number of seconds, then prints a message. Both the waiting time and the message are command line parameters. This dummy job does absolutely nothing functional, yet an advantage is that it requires minimal staging in (copying files to the clusters where they are needed). Also, we can calculate exactly how long such a job should run under optimal circumstances, meaning that given a certain number of available nodes, we can calculate how long the job would run if incurred no overhead whatsoever. The latter allows us to make fairly accurate estimates of the overhead and efficiency of KOALA-CS. All in all, the dummy application is purely intended for experiments.

2.4.3 Eternity II

The Eternity II puzzle [10] consists of a square board with 16×16 spaces for square pieces. Each piece has a pattern on each of its four sides. This pattern can be gray, identifying that piece as a border or corner piece. The goal of this puzzle is to place all of the 256 pieces on the board in such a way that the patterns on adjacent sides match. One piece is given as an anchor. Finding a solution is computationally hard.

Our goal with this application is to map it to subproblems which can be run as CS jobs using KOALA-CS. These subproblems make up our testing application. We use a random walk to create a certain setup of the board, then measure its fitness. For a number of our experiments, we use this random walk program as the target application. Based on the data thus acquired, we build our solution further, however, that process is beyond the scope of our thesis.

2.5 Problem Statement

Now that we have a clear understanding of the environment we work in and the systems we use, we define our problem statement. In this thesis, we ask ourselves this question:

> "*How do we extend* KOALA *with cycle scavenging functionality?*"

The question seems simple, yet is one with many layers. A CS extension for KOALA has a number of requirements to fulfill.

First, KOALA-CS should be *unobtrusive*. This is the predominant quality of any CS system: other, higher-priority, jobs should not be hindered by CS jobs. The system must take this into account when growing and shrinking CS jobs. When growing, KOALA-CS must allocate only those nodes to the CS job that are idle and will remain idle in the foreseeable future. When there are jobs in a local queue or KOALA's placement queue waiting for those nodes, and they could be placed, those nodes should not be allocated to a CS job. CS job shrinking should be swift. KOALA-CS must detect local and grid jobs waiting, and take action to shrink CS jobs as quickly as possible if to allow those waiting jobs to run. In desktop grids, the return of control of the desktop to the user should be near-instant. Fortunately, we can relax that requirement a little, since grid and local jobs usually have a long, non-interactive runtime. In addition, they are already delayed by the usual scheduling overhead. Therefore, we state that the additional overhead due to CS jobs is allowed to be in the order of tens of seconds. A last note on unobtrusiveness is that the CS job should not permanently affect the node it is running on. It must clean up files and reverse any other changes it has made as soon as it vacates that node.

Second, KOALA-CS must ensure *fairness*. Multiple CS jobs can be active in the system at the same time. When a grow opportunity presents itself, or job shrinking is called for, all CS jobs in the system must be fairly affected. The conclusion is that the CS jobs should all have comparable throughput (in number of tasks completed per second) when running comparable jobs.

Third, KOALA-CS must be *robust* and *fault tolerant*. Node allocations change rapidly, CS jobs generally run for a long time, and the system and environment are highly complex. Due to these reasons, KOALA-CS will encounter errors. These must be prevented as far as possible. Otherwise, they must be detected and handled properly. If they cannot be prevented or handled in any way, KOALA-CS must ensure that as many valuable intermediate results are saved as possible.

Finally, KOALA-CS must handle all of the above in an *efficient* way. Therefore, KOALA-CS must assign tasks to nodes in such a way as to minimize the runtime of the CS job. Also, the number of messages sent between components should be minimized. Shrinking must likewise be efficient, in that the CS job should be shrunk the minimal amount needed without unnecessarily delaying the waiting jobs.

Chapter 3

The Design and Implementation of Koala-CS

In this chapter we cover the design choices we made regarding KOALA-CS and how we implemented those choices. As stated in Chapter 1, KOALA-CS extends KOALA with CS functionality. In addition to the three KOALA components described in Chapter 2 (the KOALA scheduler, the runners, and the KOALA component manager (KCM)), we define a new component, the launcher. The KOALA scheduler already provides facilities for malleable jobs, with the ability to send grow and shrink messages to runners. Since the scheduler is aware of all grid jobs, it is the ideal component to govern this grid-wide malleability. The KOALA runners provide application-specific operations, such as application-level scheduling, job component management, file transfers, etc. Therefore, to handle the intricacies of CS and BoTs, and allow for application-level scheduling policies, we must create a CS runner. Furthermore, we adapt the runners framework to one specifically suited for CS jobs. The CS runner fully utilizes KCMs for component submission. In addition, we extend the KCM with the ability to grow its node allocation and with the ability to inform the runner of the need for shrinking due to local jobs. The extended KCM also deploys launchers instead of simple placeholder scripts. KOALA-CS deploys such a launcher on every node KOALA-CS uses and acts as both a placeholder and a parent process of the actual tasks for that node. This mechanism allows us more control over task scheduling and task preemption.

In Section 3.1, we discuss how KOALA-CS utilizes and extends the capabilities of the scheduler. We discuss the extended KCM in Section 3.2. In Section 3.3, we describe the launcher mechanism. As the last of the components, in Section 3.4, we present the CS runners/runners framework, since we cannot effectively describe its activities without some prior knowledge of the other KOALA-CS components. With a complete knowledge of the components of KOALA-CS in mind, we cover the job flow procotol in Section 3.5. In Section 3.6 we address the fault tolerance of KOALA-CS. Finally, we discuss the policies of KOALA-CS in Section 3.7 .

3.1 The CS Functionality of the Scheduler

In the context of KOALA-CS, the scheduler handles idle resource discovery, ensures fairness among CS jobs, and guarantees grid-level unobtrusiveness, along with regular grid scheduling. Resources usable by CS jobs are those resources that are not required by other, higher-priority, jobs, i.e., idle resources. Fairness among CS jobs requires the scheduler to divide the available resources evenly among CS jobs. Grid-level unobtrusiveness requires the scheduler to detect waiting grid jobs and find the resources for them with minimal delay due to CS jobs. We present the procedures and policies regarding resource detection, fairness, and grid-level unobtrusiveness in the sections below.

3.1.1 Resource discovery

In KOALA-CS, resource detection follows the standard KOALA process, as detailed in Section 2.2. The KOALA scheduler allocates nodes to CS jobs, which are appended to the super-low queue of the KOALA scheduler, as soon as there are no higher priority jobs that need those resources. The WRR policy (see Section 2.2.3) ensures that nodes are allocated to CS jobs only if higher priority jobs have had the opportunity to claim those nodes.

With a rigid job, as soon as there are nodes free, the job is moved from the placement queue to the claiming queue of the scheduler. However, CS jobs are malleable, and they remain in the placement queue even as parts of the job are running. This way, the scheduler will consider CS jobs for node allocation even if the job is already running on some nodes, thus allowing CS jobs to grow. In addition, that the CS job remains in the queue allows the scheduler to track the CS jobs for the purpose of shrinking them, which we discuss further in Section 3.1.3. Note that the scheduler does not track whether or not the CS job is finished. This requires the job's runner to indicate to the scheduler if the job is done, otherwise the job might stay in the placement queue forever. On the other hand, should the scheduler at some point be unable to further contact a job's runner, it will also terminate that job.

Concluding, we see that CS jobs arrive in the super-low queue of the KOALA scheduler and remain there until the runner signals that the job is done. They get assigned nodes (i.e., they are allowed to grow) as soon as nodes are free and no job in the higher priority queue requires them any longer. These nodes are divided among the available CS jobs, a process we describe further in Section 3.1.2.

3.1.2 Fair allocation of resources

The scheduler handles fairness for CS jobs because the scheduler is the only component aware of all CS jobs. It is therefore the only component that can actually

divide the nodes over CS jobs in such a way that all get a comparable number of nodes.

When nodes become available to assign to CS jobs, the scheduler must first be divide them among CS jobs to give each job a fair share of the nodes. The KOALA scheduler determines which CS jobs get allocated which nodes through a CS policy (CSP). Conversely, when there is a need to decrease the total node allocation of CS jobs, the CSP determines which CS jobs must release nodes, on which cluster they must release them, and how many they must release.

The CSP is a modular component of the scheduler. KOALA-CS may therefore run with different policies, where each uses a different methodology to provide fairness among CS jobs. However, only one CSP can be active at a time, and to switch CSPs we must reset the scheduler. Therefore, we define multiple policies in Section 3.7.1 and we test which of these policies is superior in Section 4.4.

3.1.3 Grid-level unobtrusiveness

In this section we discuss how the scheduler deals with grid-level unobtrusiveness, which is one part of the unobtrusiveness requirement of KOALA-CS (the other being unobtrusiveness towards local jobs). The unobtrusiveness towards grid jobs is the responsibility of the scheduler, since it is the only component aware of all the grid jobs, both CS and non-CS.

When jobs arrive in the higher placement queues or the low placement queue, the CS jobs (in the super-low queue) may be forced to release nodes in order to let the higher priority jobs run. This must be done in such a way that the higher priority jobs are not considerably delayed. To ensure this, the KOALA scheduler keeps track of how many nodes are assigned to each CS job and to which cluster those nodes belong. When a new, higher priority, job arrives, the scheduler calculates how many nodes must be freed up from CS jobs, and then sends the appropriate runners shrink messages. The CSP must ensure that this happens in fair way, i.e., that all CS jobs suffer equally. When enough nodes have been freed, the non-CS job that caused the shrinking can be placed.

The procedure, which we illustrate in Figure 3.1 is as follows.

1. A non-CS grid job arrives at the scheduler.

2. The scheduler looks on which cluster(s) it could place the components of that job, and determines that the entire job must be placed on Cluster 1. If it cannot place a component on a cluster because of CS jobs in the way, it determines how many nodes must be freed on that cluster. In this case, it finds that CS jobs A and B on Cluster 1 must shrink. Using its CSP, it determines how much each of the two jobs must shrink.

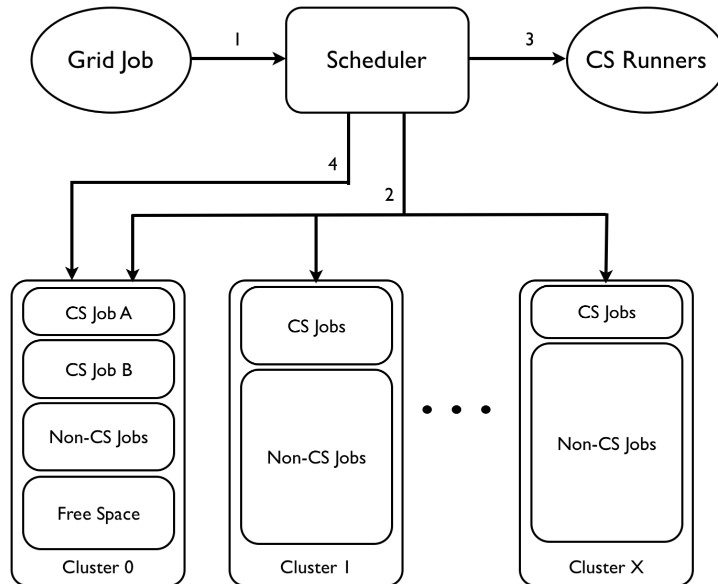

Figure 3.1: The scheduler enforcing grid unobtrusiveness

3. The scheduler sends shrink messages to the runners of CS jobs A and B, indicating on which cluster (in this case, Cluster 1), and how many they must shrink their jobs.

4. The scheduler places the new job on Cluster 1.

3.2 The Extended Koala Component Manager

The KCM plays a key role in KOALA-CS. In addition to allowing submission of tasks through the DRMAA interface of the local scheduler (as is its normal mode of operation, see Section 2.2.5), in KOALA-CS, it also detects local jobs. Therefore, the KCM supports unobtrusiveness for local jobs, i.e., it is a part of the functionality that guarantees that local jobs are delayed as little as possible by CS jobs. Furthermore, the KCM places placeholder scripts with more functionality (i.e., launchers, which we discuss further in Section 3.3) in KOALA-CS, and in doing so, plays a slightly different part in component submission than for non-CS grid jobs.

3.2.1 Local job detection

The KCM does local job detection (and thus plays a crucial role in unobtrusiveness towards local jobs) by polling the queue of the local scheduler every t seconds.

The default value for t is ten seconds, and can, with the current implementation, not be changed without recompiling the KCM. When polling the local scheduler, the KCM checks for the total number of nodes needed by jobs currently waiting, and the total number of nodes claimed by CS jobs. It can distinguish CS tasks from other tasks by checking the task name as it appears in the queue listing of the local schedulers. If there is at least one non-CS job waiting, and at least one node claimed by a CS job, the KCM sends a message with command `SHRINK` to the runner. This message contains the total number of nodes needed by waiting jobs, and total number of nodes claimed by CS jobs. It is then up to the runner to act appropriately to the situation.

By letting KCMs send messages every poll interval, the situation would be little different from the runner or scheduler polling the local scheduler regularly. And that is exactly a situation we avoid by letting the KCM poll the local scheduler. The KCM is at the cluster, located on the cluster's headnode, and can thus access the local scheduler directly. Other components in KOALA-CS would need some form of messaging to reach the local scheduler of remote clusters. Therefore, the KCM contains this local scheduler polling mechanism, and sends messages to the runner only as shrinking is needed.

In Figure 3.2 we show the situation of one cluster with multiple KCMs on it. When a local job arrives, the following happens.

1. As stated, a local job arrives at the local scheduler of a cluster, but remains in the queue because there are no nodes free for it.

2. The KCMs detect the local job in the queue and the fact that there are CS jobs active on the cluster.

3. The KCMs each send a message to their respective runners, indicating the total size of jobs waiting in the local queue, and the total number of nodes claimed by CS jobs on the cluster.

3.2.2 Component submission

The default KCM deploys placeholder scripts on nodes that send the name of their node back to the KCM, which then gathers these names and sends them to the runner (see Section 2.2.5). The extended KCM developed as part of KOALA-CS deploys launchers and then waits for those jobs to terminate. It deploys these placeholder jobs with all the information they need to run independently of the KCM. The KCM follows a likewise procedure when it receives a grow message; it simply submits the number of launchers indicated in the message. When all placeholders scripts are done, the KCM terminates.

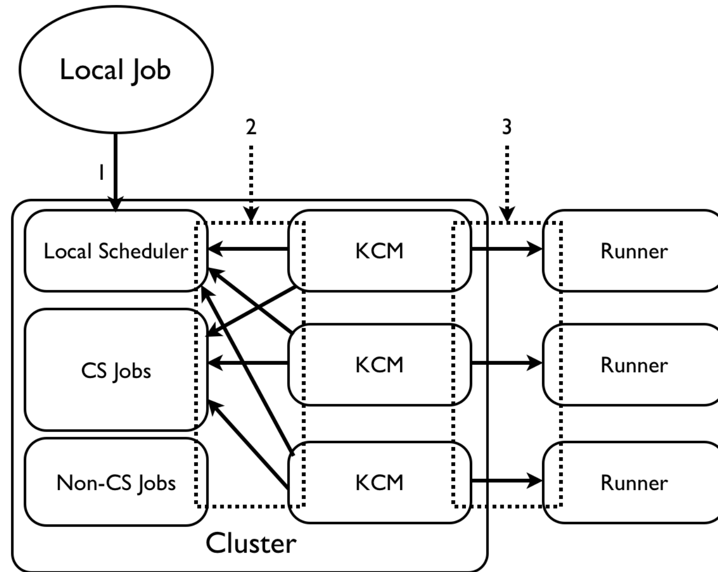

Figure 3.2: KCMs enforcing local unobtrusiveness

3.3 The Launcher Mechanism

The launcher mechanism consists of the launchers deployed by KCMs in KOALA-CS. It is a system like those used in Condor Glide-In [22] and Falkon [34]. These launchers are in themselves recognized as tasks to be executed by the local scheduler; however, they execute the actual tasks of the CS job as child processes. This way, KOALA-CS has more control over the scheduling of individual tasks, and can more easily preempt tasks and schedule new ones on the same node.

3.3.1 Deployment protocol

The launcher's controller (which is usually a CS runner) deploys launchers through a KCM, either with a new deployment of that KCM, or by growing that KCM (see Section 3.2). We illustrate the process in Figure 3.3, which shows the following steps.

1. The controller directs a KCM to deploy a launcher.
2. The KCM deploys the launcher while passing it the following parameters: the controller's hostname, the controller's port number, and the name of the sandbox directory the launcher should use.
3. Once deployed, the launcher informs the controller of the launcher's hostname and port number.

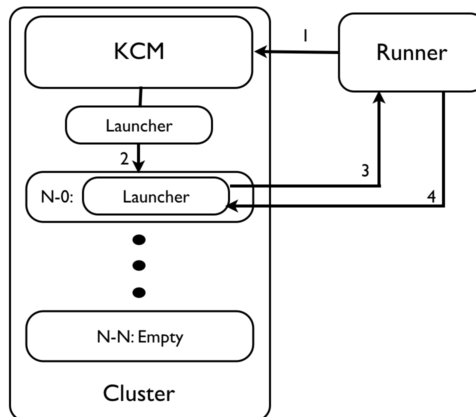

Figure 3.3: Launcher deployment

4. The controller informs the launcher of its launcher ID, with which it can further identify itself.

3.3.2 Scheduling control

The launcher allows more scheduling control because it takes the scheduling of the individual tasks that the job consists of away from the local scheduler and gives that control to the launcher's controller. Because the launchers act as placeholder tasks on the nodes, as far as the local scheduler is concerned, that node is currently in use and nothing can be scheduled there. However, the launchers themselves contain a list of tasks that they execute as child processes on the node. Between these tasks, there is none of the scheduling overhead that would occur when each task of a job would have to be scheduled by the local scheduler individually, although there is some delay as the launcher collects results and sends them to the controller.

3.3.3 Task management

The launcher has a list of tasks that it executes during its lifetime. This list may grow and shrink, and even may be empty, causing the launcher to be idle. Through messages, other KOALA-CS components may add and remove tasks from this list.

The launcher gets its original task assignment from its controller as soon as the controller becomes aware of the launcher's existence (see Section 3.3.1). After receiving this (list of) task(s), it starts by executing those task(s), assuming that all the necessary files are in the sandbox directory or otherwise available to the node the launcher is running on. It captures the standard output of the task in a file in the sandbox directory. Upon the completion of each task, it sends the controller a message indicating so, including the name of the file that contains the output of the

Command	Description
DROP	Tells the launcher to drop a task.
ID	Tells the launcher its ID, which it must include in future messages to its controller.
POLL	Polls the launcher if it is still active.
PREEMPT	Tells the launcher to terminate immediately.
TASKS	Adds a task to the launcher's task list.

Table 3.1: Message interface to the launcher

task. The launcher then removes the task from the list; if this leaves the list empty, it enters a waiting cycle.

In addition to normal task completion, a task may be terminated in two ways that do not yield results. First, a task may fail. The launcher detects a task failure only if the child process running the task returns an exit code that is not zero. In that case, the launcher sends any relevant information, including the exit code, to its controller and removes the task from its list as if completed. Second, when the launcher is preempted, any task it is running is terminated, and no further action is taken.

3.3.4 Communications

In this section, we discuss the communication between the launcher and other components of KOALA-CS. Table 3.1 shows the message interface that other components can use to control the launcher, while Table 3.2 shows the messages that the launcher sends to its controller. Messages consist of one of the commands from Table 3.1 or 3.2, a separator, and the message body. Outgoing messages are prefixed with LAUNCHER and a separator, after which they are wrapped in the standard KOALA message format, with the APPLICATION_CALLBACK command.

In Table 3.1 we list the commands that can be issued to the launcher. The ID command has been explained in Section 3.3.1. It is this message that informs the launcher of its ID. TASKS and DROP are two related, but opposite, commands. The former informs the launcher of a task to execute, the body of the message containing the actual command line, together with the task ID. The launcher than adds this task to its task list (see Section 3.3.3). The DROP command contains a task ID of a task that has to be dropped from the list. Should the launcher already be running that task, it will complete it, however. POLL is used to check if the launcher has not crashed. If it is still alive, it will reply with an ACTIVE message as described in Table 3.2. Finally, PREEMPT causes the launcher to immediately stop everything it is doing — most importantly, the task it is currently running as

Command	Description
`ACTIVE`	Indicates that the launcher is (still) active.
`ADDRESS`	Informs its controller of the host name and listening port of the launcher.
`ERROR`	Informs the controller that an error occurred while executing a task.
`FAILURE`	Informs the controller that a task failed, i.e., completed with an exit code other than zero.
`RESULT`	Informs the controller that a task was just completed, together with the task's runtime and the name of the standard output file of the task.
`STATUS`	Indicates that this is a status message.

Table 3.2: Messages from the launcher

child process —and terminate.

Table 3.2 contains the commands for messages from the launcher to its controller. These outgoing messages are a collection of informative messages, where `STATUS` is the message used to indicate anything that does not fit in with the other messages. `ACTIVE` is the proper response to the `POLL` message from Table 3.1. The `ADDRESS` message is sent as part of the launcher's deployment. With it, the launcher informs its controller of the launcher's host name and listening port. `RESULT` and `FAILURE` are two possible messages that the launcher sends as it completes a task. `RESULT` comes with the runtime needed for the task and the name of file containing the task's standard output; this message follows successful task completion. `FAILURE`, on the other hand, contains the exit code of the task, if it "completed" with an exit code other than zero (which should indicate the task failed). Finally, the `ERROR` message is a general-purpose message to indicate that some terminal exception was encountered by the launcher. It contains the error message itself, and implicitly informs the launcher's controller that the launcher has terminated due to this error.

3.4 The CS Runner and Runners Framework

The CS runner is the core of KOALA-CS. It provides all the application-level scheduling and task management functionality. It handles the needs of the KOALA scheduler and the local schedulers when those require the CS job to release nodes. In addition, it guarantees job continuity in case of a system crash (see Section3.6.3). The CS runner also handles file transfers and various other tasks. For KOALA-CS,

we extend the runners framework, as we create a version dedicated to CS jobs. This new framework allows for the creation of different CS runners with different policies, while adhering to the same protocols.

3.4.1 Structure of the CS runners framework

The CS runners framework provides an abstract base class for all CS runners, namely the `ACSRunner` class. This class enforces the job flow and communication protocol of all the CS runners. Also, like the runners framework, there is a `RunnerListener` class that acts as a message server for messages to the runner. `RunnerListener` relays the messages to the callback methods of `ACSRunner`. In order to define a different application-level scheduling policy, the runner programmer must create a subclass of the `ACSRunner` class with the new policy. In addition, the programmer may override how data and grid information are managed in the runner, how job submission to the runner works, and how to handle application callbacks that are not from a KCM or launcher. However, default functionality is provided in those cases, as opposed to the application-level scheduling, which must be defined by the programmer. The CS runners framework provides constants and easy access to several of KOALA's entity classes, such as the job and component representations. In addition, the framework provides a number of utility libraries for such things as remote execution.

3.4.2 Runner components

The runner consists of five components: the Runner Controller (RC), the User input Manager (UM), the Information Manager (IM), the Data Manager (DM), and the Application-level Scheduler (AS). Of these components, the RC is fixed because it enforces the KOALA-CS message and control flow protocols. The other components are modular, provided the implementations adhere to the correct interface. For the UM, the IM, and the DM, the runners framework provides default implementations, while the AS is runner-specific and must be provided by the runner programmer. In the remainder of this section, we discuss each component in more detail.

The runner controller

The RC enforces the control flow protocols in the runner. It also handles all messaging and component polling. Upon starting the runner, the RC enforces that any command line parameters given to the runner are parsed, including any job description files that may be passed to the runner.

The RC first creates a new runner object of the class (which must be a subclass of `ACSRunner`) that is passed to the `main` function. It then ensures that

this object first parses any command line parameters, including any job description files that may be passed to the runner. Once this is done, the RC initializes a `RunnerListener` object with the runner object as argument. It then starts this object, which creates a separate thread of control for the `RunnerListener`. Within that thread, the RC first registers the runner's job with the scheduler (through a `JOB_NEW` message, see Table 2.2), then it starts the launcher polling thread (see Section 3.6.2) and threads for active AS and UM, if they exist. Having started the `RunnerListener` thread, the RC enters a waiting loop where it regularly checks whether or not all tasks are completed. Once all tasks are completed, the RC initializes the necessary clean-up and post-termination phases, before terminating the runner.

The RC further functions as a message listener. The `RunnerListener` hands all incoming messages to the callback methods of the RC, which handles them appropriately. Every message is handled in its own thread, allowing concurrent message handling. The RC contains callback methods for messages from the scheduler, from the KCMs, and from the launchers. The RC determines which message is from which component (based on the message headers), and relays them to the appropriate handler. In addition, the RC contains a general message handler, for messages from any custom components. The default response to such messages is to discard them, but the runner programmer can override this behavior. For instance, a remote user interface can communicate with the runner through this method.

The manager components

The manager components handle different aspects of the CS job. Each manager is modular, with default versions provided for each. The UM handles command line parsing, job definition languages, and interactive user input. The default UM handles only command line parsing and the parsing of a job definition file. An advanced UM may take commands from the user at job runtime, allowing adding and removing tasks, or viewing results. The IM tracks all information regarding tasks, such as the state they are in, which launcher is running them, runtime, etc. In addition, it tracks information regarding nodes, launchers, clusters, and file locations. The IM contains functionality to make a so-called state dump to secondary storage, so the runner can be aborted at any time and restarted with its tasks in the same state, and with as little as possible loss of result data. The default implementation of the IM uses a collection of entity objects to track the data, and uses Python's cPickle [6] library to create the state dumps. Finally, the DM provides the possibility to (asynchronously) transfer files. The default DM uses secure copy (SCP) to transfer the files, and assigns each transfer a thread so as to allow asynchronous transfers.

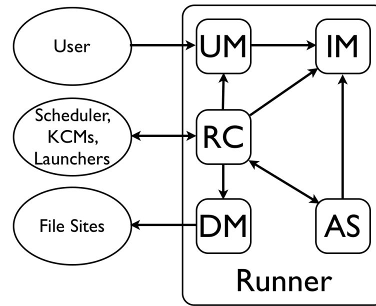

Figure 3.4: Interaction of runner components

The application-level scheduler

The AS is another modular component, for which no default implementation is provided. The runner programmer must provide his own AS implementation, which provides the application-level scheduling policy. We discuss our policy in Section 3.7.2. In its most primitive form, the AS must provide a method for assigning tasks to launchers, and for determining which launchers should preempt in case of shrinking. In addition, to AS may provide an alternate method for determining by how much to grow (up to the size allowed by the scheduler, off course), which defaults to accepting everything the scheduler offers. Finally, the AS may provide an active scheduling function, that is run as a separate thread, and may actively change the assignment of tasks to launchers, without being prompted by the RC to do so.

3.4.3 Interaction of runner components

The interaction between runner components takes place through method invocations and function calls on the various components. Central to this scheme is the RC, which governs runner activities. However, the UM and AS components may exert some control depending on their implementations. In Figure 3.4 we give an overview of the relations of the components.

Figure 3.4 illustrates the central role of the RC. The RC receives and sends all messages to and from the runner. It also starts the active parts of the UM and AS components, if any, and initiates command line parsing by the UM. Finally, it updates the IM and commands it to dump its state when required, and orders the DM to transfer any files. The UM receives any and all user input, be it from the command line or otherwise, and updates the IM with the information it thus acquires. The IM is a passive component; albeit a vital one, that stores all information relevant to the CS job. The DM again is passive; its role is to transfer files to and from the file sites, and to delete files if necessary. The AS communicates with both the IM and the RC. From the IM it acquires the information needed for scheduling. In

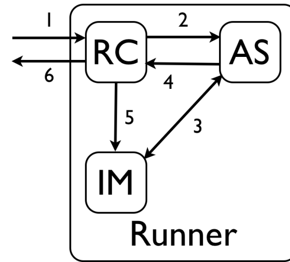

Figure 3.5: Runner response to grow and shrink messages

case of an active AS, it commands the RC to effectuate any scheduling changes the AS makes.

3.4.4 Handling grow and shrink messages

One of the dominant operations in KOALA-CS concern the malleability, or the growing and shrinking of CS jobs. In this section, we focus on the activities within the runner when a grow or shrink message arrives. In Section 3.5 we place the runner's activities within the greater perspective of KOALA-CS.

We identified two reasons for job shrinking: jobs waiting in the local queue and grid jobs waiting in one of the queues of the KOALA scheduler. Conversely, job growth is always initiated by the scheduler. Therefore, we have two messages from the scheduler (growth and shrinking due to grid jobs) and one from the KCM (shrinking due to local jobs). We illustrate the job flow for all three types of events with a single graph in Figure 3.5.

In the figure, we see six steps for each type of event. In case of a grow message, KOALA-CS proceeds as follows.

1. A grow message arrives at the RC, containing a set of cluster/nodecount pairs. This set indicates the clusters where the runner's job can grow, and how many nodes are potentially available on each cluster.
2. The RC asks the AS how many nodes to accept on each cluster.
3. The AS acquires the information that it needs for its scheduling decisions from the IM.
4. The AS answers the RC's query from step 2.
5. The RC registers the changes with the IM.
6. The RC sends the appropriate messages to other KOALA-CS components, to affect the changes.

The steps for a shrink message, regardless whether the message originates at a KCM or at the scheduler, are the following.

1. A shrink message arrives at the RC, containing the name of the cluster where the RC is to shrink (the name is implied if the message comes from a KCM) and the number of nodes to release there.
2. The RC asks the AS to determine the *shrink schedule*, i.e., set of launchers to be preempted in order to fulfill the shrink command.
3. The AS acquires the information that it needs to determine the shrink schedule from the IM.
4. The AS answers the RC's query from step 2.
5. The RC registers the changes with the IM.
6. The RC preempts the launchers in the shrink schedule.

3.4.5 Launcher management

An important activity of the runner is to manage the launchers. The runner must manage launchers, keep track of what they are doing, if anything, and if they can do what they are ordered to do.

For the deployment of launchers, the runner uses a KCM. Upon deployment, the launchers send their hostname and listening port to their controller, which is the runner. The runner logs this information in the IM, which in response supplies a unique launcher ID. The RC submits this ID to the launcher.

When launchers need to be preempted, the AS determines which launchers exactly, and the RC sends `PREEMPT` messages (see Table 3.1) to those launchers. It likewise makes the required changes in the IM. The IM ensures that any tasks that the preempted launchers were working on are updated to reflect their new state; the IM also administratively removes these launchers from their nodes.

The RC handles any information from the launchers (results, failures, errors) and stores it in the IM. This information may require further action from the runner. Errors indicate the launcher has terminated, and the IM needs to make the appropriate changes. Results and failures indicate that a task has completed or failed. The RC updates the state of said task in the IM, and, in case of results, commands the DM to start transferring any output files from the launcher's file site to the runner's file site.

When a launcher sends a message stating it has completed a task, failed at one, or has just become active, the launcher may (or, in the latter case, *does*) have nothing to do. Therefore, the runner may need to assign tasks to this launcher, or preempt it.

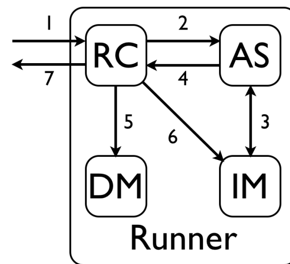

Figure 3.6: Launcher management

This requires a complex procedure which includes four out of five runner components. We illustrate this procedure in Figure 3.6. The seven steps in this procedure are as follows.

1. The RC receives a message that a launcher has a possible need for more tasks.
2. The RC asks the AS for a task assignment for the launcher.
3. The AS acquires the necessary information from the IM and makes a task assignment for the launcher, or decides it must be preempted.
4. The AS returns this decision to the RC.
5. The RC commands the DM to transfer any files needed by the launcher to the launcher's file site.
6. The RC makes the necessary updates in the IM.
7. The RC submits the new task assignment, or a preemption message, to the launcher.

3.5 Koala-CS Job Flow Protocol

In this section, we cover the five phases of a CS job's life cycle. We describe job submission, job growth, job shrinking due to grid jobs, job shrinking due to local jobs, and job completion, in the following subsections.

3.5.1 Job submission

CS job submission follows the same procedure as for any other KOALA job. Every KOALA job must have at least one component, and for that purpose, the CS runner

submits a single component that requires a single node. Upon submission of the job to the scheduler, the CS runner informs the scheduler of the properties of the job, such as user name, port number of the runner, and the type of job, which is *malleable*. We choose a size of one to make placement of the component as easy as possible. The scheduler, once it has placed this component, informs the runner on which cluster this component is placed. The runner places this component using a KCM, as described in Section 2.2.5 and illustrated with Figure 2.8. Note here, however, that the placeholder script is replaced by a launcher. Once this launcher is placed, the application-level scheduling can begin. After job submission is completed, the CS job can start to grow and shrink, until it is completed.

3.5.2 Job growth

CS job growth is initiated when the scheduler detects idle nodes, that are not needed by other (non-CS) jobs. When this happens, the scheduler divides the idle nodes among CS jobs as in Section 3.1.2. It informs the corresponding CS runners of the grow possibility with a message that consists of (a list of) cluster(s) and for each cluster, the number of nodes the runner can claim there. The runners respond by acknowledging the message. Afterwards, the runner either sends a grow message to the KCM on each cluster with the associated number of nodes, or it places a KCM on the cluster (with the number of nodes as parameter) if there is no KCM there yet. This KCM then deploys launchers as usual. We describe this procedure with Figure 3.7. The steps in the figure are the following.

1. The scheduler sends a grow message with a site s and a node count n to the runner.
2. The runner determines a number of nodes m, where $m \leq n$, to accept, and sends an acknowledgement (`ACK_GROW`) of the grow message to the scheduler.
3. The runner sends a grow message to its KCM on s for m. If there is no KCM on s, the runner deploys a new KCM there with initial size m.
4. The KCM schedules m new launchers to the local scheduler of s.
5. The local scheduler places the launchers on nodes of s.
6. Each launcher sends an `ADDRESS` message to the runner.
7. The runner responds by sending each launcher that launcher's ID, and an initial task assignment.

The described procedure concerns the activities of KOALA-CS for a grow message with a single site/nodecount pair; however, in most cases, the scheduler sends

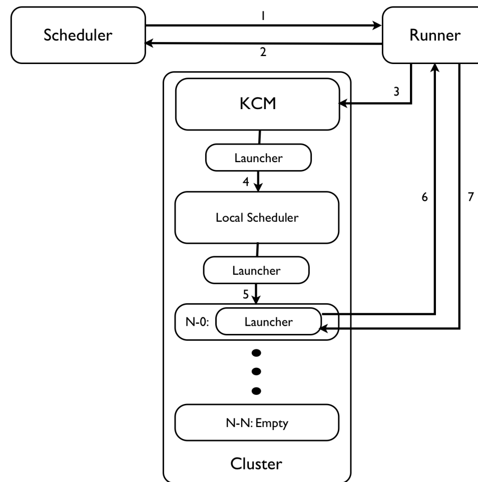

Figure 3.7: Job growth

multiple such pairs in the same message. For simplicity, we limit the message in Figure 3.7 to a single pair; however, activities for multiple pairs in a single message are essentially the same. Note that a message with multiple pairs is always answered with a single acknowledgement regardless of the number of site/nodecount pairs.

3.5.3 Job shrinking

Job shrinking occurs when either the scheduler detects grid jobs waiting (see Section 3.1.3) or the KCM detects jobs in the local queue (see Section 3.2.1). Whatever the source, the component in question sends a message to the runner, which then determines how to comply with the shrink command, depending on its application-level scheduling policy (see Section 3.4.4). We illustrate the procedure with Figure 3.8. There are two basic steps to this procedure, and these are as follows.

1. The scheduler or one of the runner's KCMs sends a shrink message to the runner. This message contains the number of nodes to free (n) and an explicit site s in case of a shrink message from the scheduler. With a message from a KCM, s is implied because the runner knows the location of the KCM in question.

2. After the runner determines a set of k launchers to preempt in order to fulfill the request for n nodes to free, or, if the runner does not have launchers on n nodes at s, it preempts all launchers on s. It sends all the launchers that have to be preempted a `PREEMPT` message.

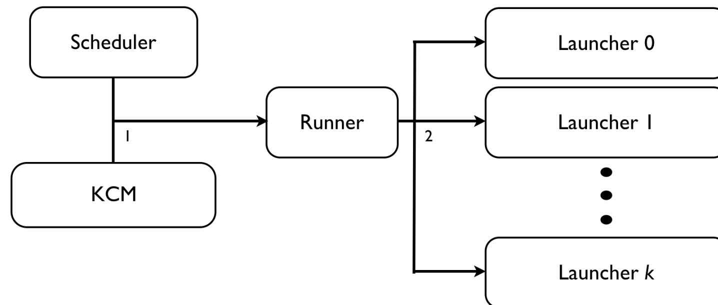

Figure 3.8: Job shrinking

3.5.4 Job completion

Job completion comes about when the runner detects that all tasks are finished. When this happens, the runner saves the results of the tasks to file, and aborts its launchers. The termination of these launchers triggers the termination of the KCMs belonging to the runner. Also, the runner informs the scheduler of the successful completion of the job.

The results of the tasks consists of the standard output files generated by the launchers (see Section 3.3.3), and the output files generated by the tasks themselves. KOALA-CS guarantees that these files have all been transferred to the runner's file site when the runner terminates. Also, the runner generates a results index file, in which, for each task, the actual command line (executable file plus parameters) is mapped to the standard output file. The runner creates this file because the task IDs the runner's IM generates are internal, and mean little to the user, while the command line does. In case the user terminates the runner before the job is completed, the runner also generates a state dump of the tasks in its IM. This allows a later restart of the runner from that state file, so tasks are not needlessly completed more than once.

3.6 Koala-CS Fault Tolerance

KOALA-CS needs to be robust and fault tolerant, reducing as much as possible the likelihood of failures and their impact on the system. In this section, we deal with those issues. First, Section 3.6.1 deals with failures prevention. Second, in Section 3.6.2, we discuss how KOALA-CS detects and handles the failures of components. Finally, we discuss how KOALA-CS can recover from system crashes in Section 3.6.3

3.6.1 Error and failure prevention

There are a number of reasons a system, especially one of the complexity of KOALA-CS, can fail. We seek here to prevent two: *programming errors* and *system overloading*. How KOALA-CS deals with other failures is the topic of the following two sections.

We reduce the impact of programming errors through the normal methods of program testing. For KOALA-CS, this means we unit test what can be unit tested (such as the IM) and test other code (for instance, the inter-component communications) by executing a reduced version of the system that emphasizes the functionality of that specific piece of code. Finally, our experimental runs are incremental: we run the least complex (e.g., those that require the least functionality) experiments first, so we can be sure at least that functionality works correctly. By using this incremental approach, we can detect the majority of programming errors, and catch those that would cause the system to crash under normal operating conditions.

System overloading is a term we use to indicate the concurrent claiming of so many nodes that the grid as a whole crashes, which is of course something we need to prevent at all costs. To do this, the scheduler enforces limits on how many nodes can be claimed for CS, such that the grid is not filled up entirely.

However, as we mentioned at the beginning of this section, there are those errors that we cannot (reasonably) prevent. With such failures, components will crash. To deal with this, KOALA-CS must detect such component failures and act appropriately to minimize their impact. Failure detection and handling is the topic of the next section.

3.6.2 Component failure handling

Since KOALA-CS uses four types of components, and each could potentially fail, KOALA-CS must be prepared to deal with this issue. Not all failures of components are catastrophic however, and for some failures explicitly dealing with them is not necessary. Note that the measures we take are sometimes drastic; this is to absolutely ensure unobtrusiveness.

Koala-CS scheduler

The KOALA scheduler has proven itself to be a highly reliable component [29], that has yet to crash since its initial deployment in 2005. A theoretical problem — which we are yet to encounter — is that if the scheduler crashes, and there are any CS runners active, they will no longer receive grow or shrink messages from the scheduler after restart, since it is unaware of their existence. This may cause the grid to be heavily loaded with CS jobs, a load which can only be reduced by local job submissions or launcher crashes. Needless to say, this will drastically reduce

unobtrusiveness towards grid jobs. One potential solution is the runner polling the scheduler for activity, and terminating itself if it cannot confirm that the scheduler is active. Another solution, as presented by Mohamed [29], is for the scheduler to write the content of its queues to disk at regular intervals, allowing the scheduler to restart with full knowledge of which runners are active. Since a scheduler crash is an extremely rare occurrence, we leave determining which is the better method, and implementing either, as future work.

CS runners

A runner crash could be catastrophic to the system. Of key importance is protecting the information about tasks, which we discuss in Section 3.6.3. The launchers will detect the runner's crash because of connections from them to the runner being refused, or because of runner time-out, i.e., when a launcher does not receive a message from the runner for more than two minutes. When a launcher detects this, it terminates as if having received a `PREEMPT` message (see Table 3.1). Eventually, this causes all launchers to terminate, which in turn causes all KCMs to terminate. In that way, all processes are neatly cleaned up, although sandbox directories will need to be deleted manually.

KOALA **component managers**

A KCM crash is another rare happening, which we did not encounter during testing. The runner will detect the KCM crash as it tries to send a grow message to the KCM and the connection is refused. This will cause the runner to deploy a new KCM in place of the crashed one. Also, the runner will check whether or not the KCM is alive on a regular basis. It does this by simply connecting to the KCM, and if the connection is refused, it considers the KCM to be crashed, and replaces it with a new one. Since KCM crashes are rare, we did not implement an explicit polling system as with the launchers.

Launchers

Finally, KOALA-CS detects launcher through the runner tracking the last message received by a launcher. If a launcher has not send a message in a while (which can be common with large tasks) the runner sends a `POLL` message (see Table 3.1). If a launcher does not reply to such a message by means of a `ACTIVE` message (see Table 3.2), the runner registered this as a launcher time-out in the runner's IM. In addition, if a connection to a launcher is ever refused, it is likewise registered. The runner responds by sending a `PREEMPT` message to make sure the launcher is indeed terminated, and updates the status of that launcher's task in the IM.

3.6.3 System failure handling

KOALA-CS, like any other software system, cannot prevent all crashes. Because of the lengthy nature of CS jobs, this poses a significant problem, as potentially months of results may be lost. Therefore, key information must be regularly saved to secondary storage, in stead of being stored only in primary memory. Fortunately, all information relevant to the job itself is stored by a single component: the runner. The RC component of the runner regularly forces a state dump of the information stored in the runner's IM. The IM component can be restarted from that state dump. This state-dump only contains information about the tasks themselves, because information about launchers, nodes, clusters, and file locations may be invalid after the state is dumped, and is not necessary to protect the results of the tasks. With this scheme, we can prevent significant loss of results data, even as the results gathered between the last state dump and the crash are lost. The interval for saving the state is defined by the runner's UM; the standard UM allows the user to change the interval, but defaults to 150 seconds, so as not to burden the system too much.

3.7 The Policies of Koala-CS

In previous sections, we focussed on the architectural design of KOALA-CS. However, in those sections we already refer to the policies of KOALA-CS. In this section, we cover those policies in detail. We first discuss the CS policies in Section 3.7.1. Second, we discuss application-level scheduling policies in Section 3.7.2.

3.7.1 CS policies

The CS policy (CSP) needs to provide fairness among CS jobs. This means that, should there be multiple CS jobs in the system, the CSP assures that the total number of idle nodes is divided equally amongst the CS jobs. With our policies, we opt for those that do not require historical information, which would require tracking such information for CS jobs. We prefer simple policies that are easy to implement and test in favor of more complex policies. We deem this sufficient because of the best-effort nature of CS, where node allocation is very irregular, and historical data and predictions may not provide so much more efficiency as to validate the added complexity.

We have created two policies based on the well-know EquiPartition policy [28]. This policy takes all the resources available, and allocates them such that at each period in time, all jobs have an equal allocation of nodes. There is a possibility to re-evaluate the current resource allocation at fixed intervals, possibly requiring redistribution. However, our policies only evaluate the situation as soon as either

there are more idle nodes to allocate (i.e., grow opportunities for CS jobs) or there is a need to shrink CS jobs.

The first policy is Grid-EquiPartition (GEP). GEP considers all the idle nodes, regardless of which cluster they are on, as a single set that is to be divided amongst the CS jobs. GEP simply divides this set evenly over the CS jobs in the system; it does not consider to which clusters these nodes belong and therefore may assign one CS job nodes that are all on the same cluster, and another CS jobs nodes that are spread throughout the system. GEP does not grow any job if the total load of the grid is 80% or higher, to prevent overloading the grid. Shrinking is site-based, i.e., a shrink request is tied to a specific cluster. The policy checks which CS jobs are active on that cluster and orders each to shrink an equal amount in order to satisfy the shrink request.

The second policy is Site-EquiPartition (SEP). SEP considers each cluster separately, and divides the set of idle nodes on each cluster over the CS jobs. Like GEP, SEP does not grow when there is a load of 80% or higher, but SEP measures it per site: it does not grow on a cluster if that cluster is loaded for 80% or more. For shrinking, the SEP policy acts the same as GEP.

In Figure 3.9 we illustrate a typical allocation of nodes to different CS jobs. We see that GEP allocates nodes regardless of cluster, while SEP divides the CS jobs evenly over all clusters. The advantage of the SEP approach is that all jobs can benefit from faster or more reliable hardware, and are all equally disadvantaged by slower or less reliable hardware. On the other hand, GEP is somewhat simpler to implement.

3.7.2 Application-level scheduling policies

In this section, we cover the application-level scheduling policies, which can be based on a variety of job, task, and grid properties, among other things. The ALS policy is implemented in the modular AS component of the CS runner. Note that these policies therefore only deal with the tasks and launchers of one specific runner. Other CS runners in the system may have different policies. The policy consists of two components: the Task Scheduling Algorithm (TSA) and the Launcher Preemption Policy (LPP).

The TSA handles the assignment of tasks to launchers. We can apply any of a number of scheduling policies, such as WQR [19], MQD [26], Backfilling [21], or XSufferage [16]. Furthermore, the runner's AS component may provide an active scheduling thread, which can re-evaluate the current schedule and redistribute tasks if needed. However, like with our CS policies, we prefer a simple TSA, which we call Single task Launcher Pull (SLP). When a launcher needs a task (it is either just deployed or just completed its previous task list), SLP assigns that launcher a random task from amongst the currently uncompleted, unassigned tasks. If no such task exists, SLP indicates that the launcher should be preempted. SLP is a

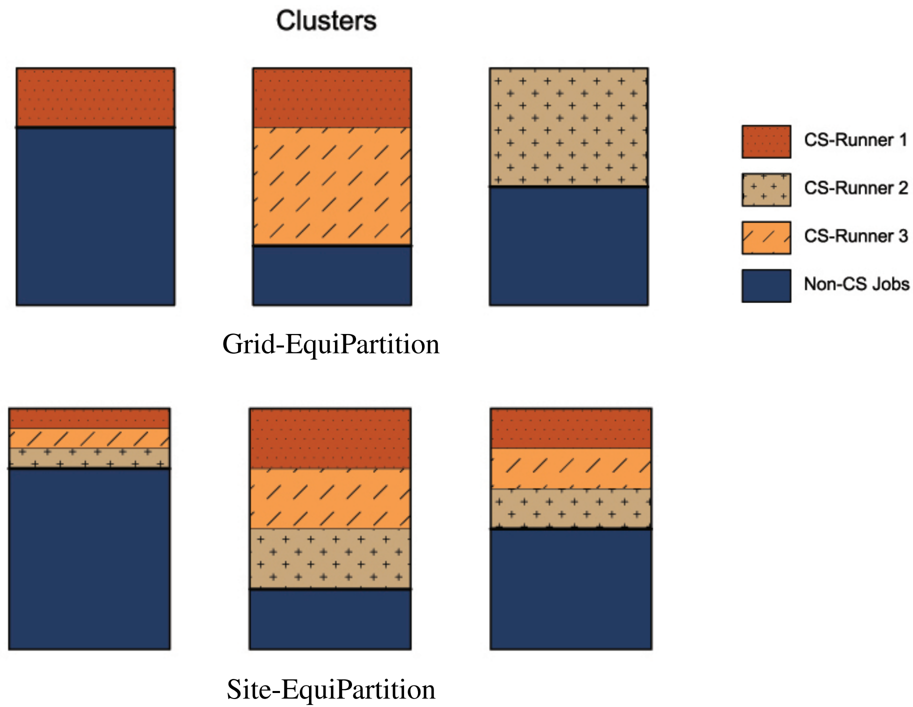

Figure 3.9: Node allocation with GEP (top) and SEP (bottom) policies

deliberately simple TSA, which facilitates the testing of the system. We leave the design, implementation, testing, and comparison of other TSAs as future work.

The LPP responds to shrink commands. As parameters, it receives a number of nodes to release and a cluster where these nodes are to be released. It converts these parameters into a list of launchers that have to be preempted in order to fulfill the shrink command. If the LPP cannot completely fulfill the node demand, it will indicate that all launchers on that cluster should be preempted. Like with TSAs, a variety of LPPs are possible. The LPP must match with the TSA, and therefore will base decisions on the same information as the TSA, in order to complement rather than obstruct the TSA. LPPs can be based on runtime of tasks, the task loads of launchers, the number of launchers running the same task, the number of times a task was preempted, etc. For our initial LPP, we use the converse of our TSA: we preempt a random set of launchers on the indicated cluster. Again, this provides us with a simple policy, and we leave more complex LPPs as future work.

Chapter 4

Evaluation of Koala-CS

In this chapter, we evaluate KOALA-CS through experiments in a real grid system, the DAS-3. We test whether or not KOALA-CS satisfies the requirements stated in Section 2.5, namely if it is unobtrusive, fair, efficient, robust, and fault tolerant. We did not evaluate robustness and fault tolerance in a separate experimental set; rather, we used all the experiments to identify and correct errors and bugs, hence improving those aspects. We ran three sets of experiments and we ran those experiments that require less KOALA-CS functionality first. This allowed us to use the experimental results directly during development.

The first experimental set assesses the efficiency of the launcher mechanism. We discuss this experiment in Section 4.2. The second set concerns the unobtrusiveness requirement. We tested for unobtrusiveness towards local and towards grid jobs; in Section 4.3, we present our findings. In Section 4.4, the final set assesses the fairness of the CS policies, by measuring throughput and node preemptions.

4.1 Methodology and Metrics

Our general methodology for testing KOALA-CS is as follows. We run our tests in the DAS-3 grid. Depending on the type of experiment, we can limit it to use only a subset of the clusters in the DAS-3. This can be useful if we need to guarantee that certain conditions are in place. Using the DAS-3 has the advantage of testing KOALA-CS in a real grid. However, changing conditions in the grid enforce the need for closely monitoring certain tests, as changing conditions may invalidate any comparison between different experiments. Likewise, the real, live, nature of the DAS-3 may cause incidental, extreme, results, which may influence the outcome of the experiments and could make them non-representative; we therefore take appropriate measures to reduce the impact of these results. Which measures we need are experiment-specific and therefore we discuss them in the section of the experiment in question.

We use two metrics: *makespan* and *throughput*. We define the makespan of a job as the difference between its submission time and the completion time of the last task of that job. We measure it in seconds, unless otherwise stated. Our definition of throughput is the number of tasks completed per time period. We measure it in tasks per second, again unless otherwise stated.

4.2 The Efficiency of the Launcher Mechanism

In this section, we discuss a set of experiments that determines the efficiency of using launchers. With these tests, we aim to validate the launcher mechanism. To do this, we compare the runtime of a set of tasks when submitted through launchers with the runtime when submitted directly to the local scheduler.

4.2.1 Experimental setup

We execute the launcher test on the 68-node Delft cluster of the DAS-3. We do this because we simply compare performance between using launchers and direct submission through the local schedulers. Using only one cluster makes it easier to ensure the same conditions for both methods, since it is easier to find a cluster that is free of background load than a period during which the entire grid is free. Also, it would make the testing scripts more complex, only to make *both* methods finish faster, which, considering these experiments do not have a long runtime anyway, we find to be an unneeded trade-off. We run the experiments when the Delft cluster has *no* background load, and mitigate any extreme results by running each experiment five times and taking the average values of the results.

We run our experiment with the dummy application described in Section 2.4.2. The experimental job consists of 1000 tasks of the dummy application, set to run for 30 seconds. In the ideal case, the makespan of this job on the Delft cluster would be $(1000 \times 30)/68 \approx 441.177$ seconds. However, the makespan must be a multiple of 30 seconds, since the dummy tasks are atomic. Therefore, the ideal makespan is 450 seconds. This means an ideal throughput of $1000/450 = 2.22$ tasks completed per second. We measure, for both submission techniques, how much it deviates, in both makespan and throughput, from the ideal case.

4.2.2 Results and discussion

We summarize the results in Table 4.1. Table 4.1(a) contains the comparison between the makespan of the direct submission and launcher submission methods with the ideal makespan. In Table 4.1(b) we compare the throughputs (in number of tasks completed per second) in a similar fashion.

Method	Makespan	Deviation from Ideal
Ideal	450.00	-
Direct Submission	685.01	152.2 %
Launcher Submission	474.85	105.5 %

(a)

Method	Troughput	Deviation from Ideal
Ideal	2.22	-
Direct Submission	1.46	65.7 %
Launcher Submission	2.11	94.8 %

(b)

Table 4.1: Performance comparison between launcher submission and direct submission.

From Table 4.1 we find that launcher submission is far more efficient, and approaches the ideal makespan to within 6%. The explanation for this is that direct submission requires, for each task, to detect an available node and then submit the task, which creates considerable overhead. Arguably, this can be reduced by waiting, per task, for it to finish, and then submit a new one the moment the old one is done. However, this still not completely eliminates the overhead of submitting a task to the local scheduler and it being queued in the local scheduler. Also, the KOALA scheduler is responsible for detecting and assigning free nodes, and the jobs that we run with KOALA-CS are CS jobs. CS jobs reside in the lowest placement queue (see Section 2.2.3), and thus can be made to wait for free nodes even more than during our experiments. Launchers, once submitted, stay in place until preempted. They thus all but eliminate the time between one task finishing and the next starting on the same node.

4.2.3 Conclusion

The results presented in Section 4.2.2 show that the use of the launcher mechanism increases efficiency. Using launchers increases the throughput and decreases the makespan of jobs. Since there is not a significant amount of additional work in using a launcher system, we believe that this validates the use of the launcher mechanism.

4.3 The Unobtrusiveness of Koala-CS

In this section, we discuss the testing of whether or not KOALA-CS is unobtrusive. We do this by measuring the additional delay that grid and local jobs incur when CS jobs are "in the way". As stated in Section 2.5, this delay must be measurable in tens of seconds, and the aim of this experiment is to show that KOALA-CS limits the delay to less than a minute. We first discuss the tests for local job delay in Section 4.3.1. In Section 4.3.2 we discuss the grid job tests.

4.3.1 Local job delay

In this section, we discuss the unobtrusiveness tests for local jobs.

Experimental setup

We define six workloads, each consisting of jobs running the dummy application from Section 2.4.2. Each job consists in total of 40 dummy tasks, which each run for 60 seconds. The jobs are however submitted in a different fashion. With each job, we divide the tasks over several atomic "chunks", i.e., the tasks are submitted such that they appear to be a task that requires multiple nodes at once, on the same cluster. We use jobs of consisting of 1 chunk of size 40, 2 of size 20, 4 of size 10, 10 of size 4, 20 of size 2, and 40 of size 1. We run our tests on the Delft cluster of the DAS-3. We choose a total size of 40 to guarantee the job will run as fast as possible, because we cannot guarantee that the cluster we test on is completely empty. Using a total size of 40, we allow room for considerable background load if we run the test on the Delft or Vrije Universiteit clusters, giving us more flexibility in planning the experiments. We run each job a total of 10 times, 5 times without any CS load and 5 times with a CS load that fills the cluster almost completely, and then we take the average in both cases.

Results and discussion

In Table 4.2 we summarize the results. The table contains, for each job, the makespan without CS jobs active and with CS jobs active, as well as the difference between the two.

From the table it is apparent that the difference in makespan for the same job under different conditions varies between 7.9 and 29.2 seconds, which is well within the acceptable range. An additional 30 second delay may seem like a lot compared to the runtimes of these jobs, however, most jobs run on a grid actually have longer runtimes. When jobs have runtimes of 15 minutes or more, 30 seconds of overhead is easily amortized. A possible source of delay is that the KCM checks the local queues every 10 seconds, and in the worst case, will detect local jobs in

Job	Without CS load	With CS load	Difference
1×40	67.932	83.058	15.126
2×20	66.696	95.931	29.235
4×10	68.586	92.827	14.241
10×4	76.076	94.038	17.962
20×2	82.477	94.978	12.501
40×1	90.188	97.975	7.887

Table 4.2: Makespans (in seconds) of the test jobs with and without CS load.

the queue while those have been there almost 10 seconds. Reducing this polling interval could potentially reduce the overhead further. Therefore, we conclude that with local jobs, KOALA-CS is sufficiently unobtrusive.

However, we also conclude from Table 4.2 that finer grained jobs suffer less from a CS load than coarser grained ones. When there is no CS load, makespans vary between approximately 67 and 90 seconds, where the finer grained jobs take longer. When a CS load is in place, the difference in makespans is less pronounced: from around 83 to around 98 seconds, and only one makespan is shorter than 92 seconds. We attribute this to a form of scheduling ”pipelining”, i.e. that local scheduling and CS task preemption overlap.

In Figure 4.1, we present an abstract overview of what happens. It shows the makespan of individual chunks of a 4×10 job in Figures 4.1(a) and 4.1(b) and the makespan of a job consisting of one chunk of size 40 in Figures 4.1(c) and 4.1(d). For all figures, the submission of a chunk occurs at time 0, and chunk completion occurs at the end of the bar. The colors indicate different stages in a chunk's lifetime: blue indicates waiting time, green indicates the time needed to preempt CS in order to make room for the chunk, orange indicates the time needed by the local scheduler to schedule the chunk, and red is the actual runtime of the chunk. Note that the makespan of a job runs from the submission of its chunks to the completion time of the chunk that completes last.

In Figure 4.1(a) we see that each chunk incurs scheduling overhead (in orange) and that while one chunk is being scheduled, the others must wait (in blue). The result is that the final chunk must wait for all the chunks to be scheduled before it can run (in red). Figure 4.1(c) shows that the scheduling time of the single-chunk job is comparable to that of a single chunk in the Figure 4.1(a). Since the runtime is the same, this larger job will complete at approximately the same time a the first chunk of the 4×10 job. However, as we consider the situation with a CS load, we get the figures as in Figure 4.1(b) and in Figure 4.1(d). As is clear from Figure 4.1(b), the scheduling of the first chunk can begin as soon as only 10 nodes are free, and while this chunk is being scheduled, node preemption for the second chunk already begins, causing these phases to overlap. The result is that the fourth chunk can be scheduled as soon as 40 nodes are preempted, because

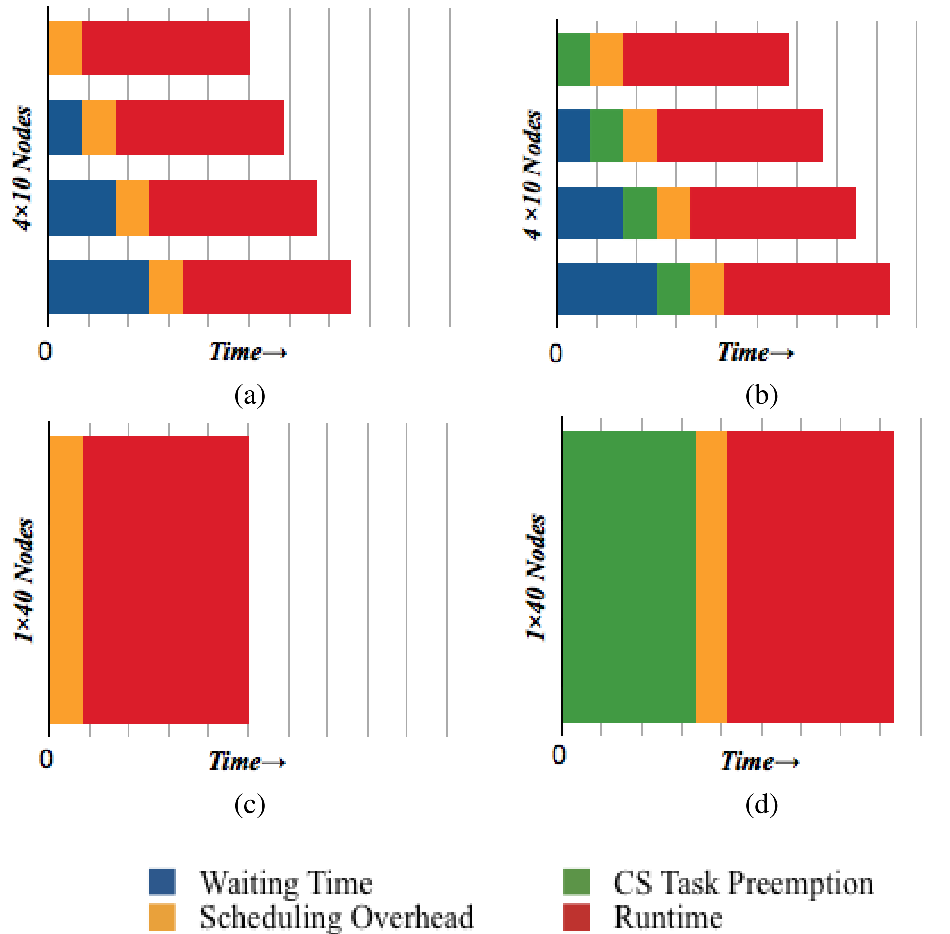

Figure 4.1: The effect of scheduling pipelining without CS ((a) and (c)) and with CS ((b) and (d)).

the other chunks have already been scheduled. The single size 40 chunk from Figure 4.1(d) must also wait for all 40 nodes to be preempted, because it needs them all, before it can be scheduled. There can be no overlap between scheduling and node preemption in this case, causing the job consisting of the single large chunk to have the same makespan as job consisting of four smaller chunks while there is CS load.

4.3.2 Grid job delay

In this section, we discuss the unobtrusiveness tests for grid jobs.

Job	Without CS load	With CS load	Difference
4×20	91.425	99.919	+8.494
8×10	87.533	87.428	-0.105
16×5	141.150	149.387	+8.237

Table 4.3: The additional delay incurred by grid jobs due to CS jobs, in seconds

Experimental setup

For this experiment, we define three jobs, each of total size 80. This total size ensures that in most cases, the load should be divided amongst clusters, since only the Vrije Universiteit cluster of the DAS-3 can provide 80 nodes on its own. All of the jobs are again dummy jobs, which run for 60 seconds (per task). Also, like the local job tests, we divide them into atomic ”chunks”. We run jobs of 16 chunks of size 5, 8 chunks of size 10, and 4 of size 20.

Since the DAS-3 is a dynamic environment, we once again run each experiment 10 times, 5 times with CS jobs active in the system and 5 times without. We alternatively run jobs without CS load and with CS load, so as to make sure that the experiments run in as similar circumstances as possible. This CS load is such, that it fills the DAS-3 up almost completely. Afterwards, we take the average of the values obtained.

Results and discussion

We collect the results in Table 4.3, which contains for each job the makespan without a CS jobs load and with such a load, as well as the difference between the two. From the results, it is clear that the additional delay cause by the CS load is negligible. Less than 10 seconds of additional delay is well within the acceptable range of one minute. The result of the 8×10 run is especially good, with the difference being *in favor* of the job ran with CS load.

4.3.3 Conclusion

Both sets of experiments show that KOALA-CS meets its unobtrusiveness requirement. The largest additional delay does not exceed 30 seconds, which is not substantial in grids. We are therefore confident that KOALA-CS is unobtrusive.

If there is any improvement to be made, we believe it lies with the local shrinking. Increasing the frequency with which the KCM polls the local scheduler will decrease the time that local jobs wait due to CS jobs, however, it will increase the processor load on the head node of the cluster in question. There may also be a minor improvement in local job unobtrusiveness by modifying the local shrink algorithm. We leave determining the ideal local scheduler polling frequency and the

ideal local shrinking algorithm as future work.

4.4 The Effect of CS Fair Sharing Policies

With the fairness tests described in this section, we aim to determine which of the two CS policies (CSPs) defined in Section 3.7.1 provides the most fair allocation of nodes to CS jobs. In order to test this, we test the performance of multiple CS runners running identical jobs over the same period, where during each period we run scripts that enforce significant background load. These loads will trigger grow and shrink events, causing the CSP to cope with these.

4.4.1 Experimental setup

The application we use in these experiments is the Eternity II application described in Section 2.4.3. We define a job that consists of running 10,000 instances of the Eternity II solver program, to guarantee that each job would need more than an hour to run even if there are no other jobs running on the DAS-3. We let three CS runners each run a copy of this job, and compare their throughputs, number of launcher time-outs, and the number of node preemptions. Combining this information, we can create an accurate view of the fairness of CS policies.

We run each of the two policies, Site-EquiPartition (SEP) and Grid-EquiPartition (GEP). We test these policies with the following three workloads, which each consists of approximately an hour of runtime.

1. *Free*. This workload consists of the CS jobs we run and any local jobs in the grid at the time of the experiment. We do not submit nor terminate any of these local jobs.
2. *Block*. In addition to the Free workload, we submit dummy jobs through KOALA. We submit these jobs as follows. We divide the experiment's runtime in three twenty-minute periods. At a random moment within such a period, we submit a grid job that requires a hundred nodes for a period of ten minutes. This workload simulates large, constant loads of grid jobs.
3. *Spike*. Similar to the Block workload, only we divide the experiment's runtime into ten minute periods, and during each period we submit a grid job that requires a hundred nodes, yet for only one minute. This workload simulates a more dynamic load of high "spikes" of grid activity.

We cannot guarantee the exact same conditions in the DAS-3 for each experiment. Neither can we guarantee the exact shape that the background load will take. There is simply too much non-determinism in the DAS-3 for that. However, since we compare the results of three jobs that run at the same time, we can draw

accurate conclusions about how fairly these jobs are treated by the CSP, and then conclude whether the policy is fair enough, and which policy is the most fair of the two.

4.4.2 Results and Discussion

In this section, we first present an overview of the load of the DAS-3 for each experiment (see Figure 4.2). Here, we see the DAS-3 load ratio on the vertical axis, as a function of time. The orange area depicts the load due to CS jobs. The blue area is the load due to KOALA jobs and the green area consists of any local jobs active in the DAS-3 at the time. These areas are stacked, meaning that the top of the orange area is the total aggregate load at that time in the system. The left column consists of the tests with SEP, while the right column consists of GEP experiments. The top row are run with the Free workload, the middle row with the Block workload, and the bottom row with the Spike workload.

Immediately apparent from Figure 4.2 are the many spikes at the top of the graph. Of course, the top of the brown area is the aggregate load of the system at that time, and therefore shows more spikes if the background loads has more spikes. However, the pattern of spikes of the orange area does not exactly match the pattern of the combined green and blue areas, most noteworthy in Figure 4.2(b), where the green area is almost constant, but the top of the graph is very ragged. We contribute this to launcher time-outs. In a dynamic grid system, failures are not uncommon, and the launcher time-out count represents those failures. A more complex launcher polling system may alleviate this problem somewhat, however, would also require computation time and memory for the runner. We therefore deem the current solution sufficient and leave a more complex system as future work; even more so since CS jobs already deal with a large number of preemptions.

In Table 4.4 we show the throughput, the number of launcher time-outs, and the number of preemptions incurred during the SEP experiments. We see a regular pattern here, where the worst case difference in throughputs is 18.6% of the throughput of the slowest job, see Table 4.4(c). In addition, we see that the number of preemptions that each job incurred differs at most by 17 (see Table 4.4(a)). We also note that in the cases where the throughput differs relatively much, the number of launcher time-outs of the slower job is usually considerably higher. Taking all of this into account, we believe that SEP indeed provides fair allocation to all CS jobs. Interesting is the high number of preemptions in Table 4.4(a), since there was no artificial background load in that experiment. However, we do see a rather regular pattern of small bursts of background jobs in Figure 4.2(a), which shows the corresponding grid load. This pattern frequently creates the need for local shrinking, and therefore a large number of preemptions. Once again, like in Section 4.3.3, a less drastic local shrinking policy might reduce this number. Also, synchronization between the local scheduler, the KCMs, and the runner plays a significant role

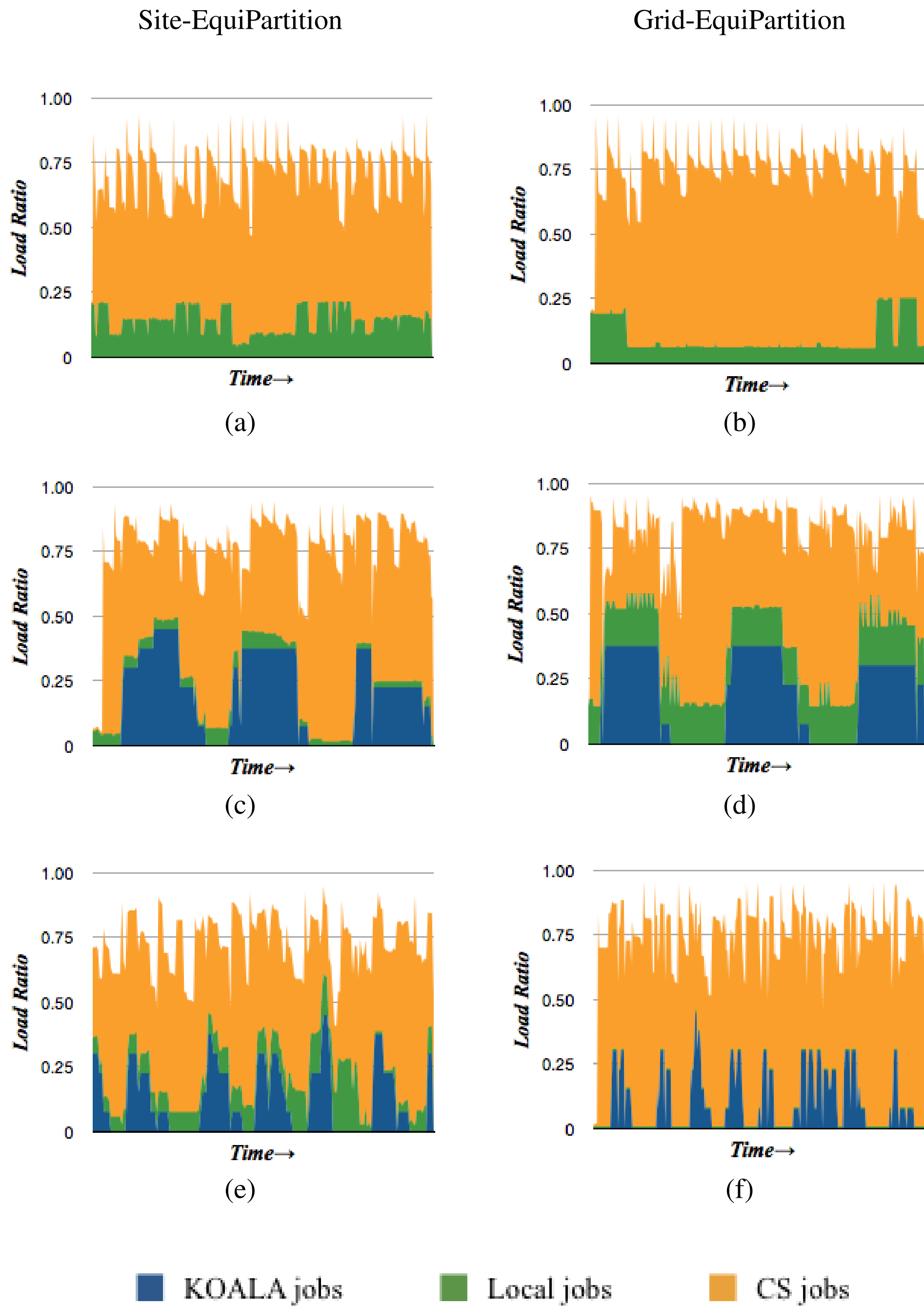

Figure 4.2: The grid load during the fairness experiments: without background load (top), with block background load (middle), and spike background load (bottom).

Runner	Throughput (tasks/s)	Number of Launcher Time-outs	Number of Preemptions
1	0.0413	85	636
2	0.0405	46	658
3	0.0449	40	653

(a)

Runner	Throughput (tasks/s)	Number of Launcher Time-outs	Number of Preemptions
1	0.0435	36	323
2	0.0399	47	321
3	0.0471	37	333

(b)

Runner	Throughput (tasks/s)	Number of Launcher Time-outs	Number of Preemptions
1	0.0427	33	568
2	0.0391	39	564
3	0.0360	47	565

(c)

Table 4.4: The performance of the Site-EquiPartition policy with (a) no artificial background load, with (b) block background load, and with (c) spike background load.

Runner	Throughput (tasks/s)	Number of Launcher Time-outs	Number of Preemptions
1	0.0858	167	119
2	0.0366	87	507
3	0.0258	16	588

(a)

Runner	Throughput (tasks/s)	Number of Launcher Time-outs	Number of Preemptions
1	0.0280	42	346
2	0.0546	4	304
3	0.0322	103	333

(b)

Runner	Throughput (tasks/s)	Number of Launcher Time-outs	Number of Preemptions
1	0.0438	50	525
2	0.0336	147	470
3	0.0644	109	309

(c)

Table 4.5: The performance of the Grid-EquiPartition policy with (a) no artificial background load, with (b) block background load, and with (c) spike background load.

in this; when a KCM signals the runner to shrink, its response will always be delayed, and the local scheduler will detect that response even later, which may cause the KCM to send another shrink message to the runner because it stills sees jobs waiting in the local queue. Increasing the time between subsequent checks by the KCM of the local queue may alleviate the problem somewhat; however, this will hinder unobtrusiveness, which is a more pressing concern. Determining the ideal polling interval and designing a more subtle local shrinking policy are therefore left as future work.

Table 4.5 shows the the throughput, the number of launcher time-outs, and the number of preemptions for the GEP experiments. We see a drastic difference with the SEP experiments. In the most dramatic case, the throughput of one runner is almost triple that of another runner in the same experiment, see Table 4.5(a). Here we see the effects of one runner having all its launchers on a cluster with relatively little background activity, while other runners are allocated nodes on more dynamic

clusters. The jobs on the ”empty” cluster then get relatively more throughput than the ones on the dynamic clusters. Even with a constant background load, GEP may yield different throughputs for different jobs, because of grid heterogeneity. With GEP, some jobs will be scheduled on slower resources then others, instead of an even distribution of jobs over all types of resources. Therefore, we conclude that fairness is not guaranteed with Grid-EquiPartition.

Obviously, with a more regular non-CS load, fairness can more easily be enforced. Also, the launcher time-outs do not always seem to have an extremely damaging effect on throughput, which is due to the fact that launcher time-outs can occur during any moment of a launcher's lifetime. If the majority occurs during a period where the launcher has no work, these time-outs may not cause task restarts, and thus have less effect on throughput than time-outs occurring while the launcher is running a task. The same holds, of course, for preemptions.

4.4.3 Conclusion

In conclusion, we see that SEP is certainly the policy that guarantees the more fair operation of KOALA-CS. Comparing GEP to SEP may seem unfair in light of, for instance, the case of the GEP experiment without background load; however, the SEP policy always allocates a roughly equal number of nodes of each cluster to each runner, and therefore each runner would suffer or benefit equally from the behavior of local jobs on that node. We see that the best performance of GEP (see Table 4.5(b)) is inferior to the worst performance of SEP (see Table 4.4(c)). We believe further improvements can be made by increasing the ideal KCM local queue polling interval, by improving the local shrinking policy, and by creating a more complex policy for polling the launchers. Most of these improvements require trade-offs in other fields, such as scalability and robustness, as well as unobtrusiveness. We therefore leave these studies as future work.

Chapter 5

Conclusions and Future Work

In this thesis, we have developed, implemented, and tested (on a real multi-cluster testbed) KOALA-CS: a Cycle scavenging (CS) system for multi-cluster grids. KOALA-CS extends the reliable KOALA grid resource manager, and requires no additional modifications to local schedulers or other grid components. We designed and implemented extensions to three KOALA components: the scheduler, the runner and the KOALA Component Manager (KCM). In addition, we designed an additional component, the launcher. Together, these four components provide unobtrusive, fair, efficient, fault tolerant, and robust facilities for CS jobs, while also being extensible for a variety of applications. With our experiments, we have shown that KOALA-CS is unobtrusive to both grid jobs and local jobs. While the non-CS grid and local jobs do incur some delay, we have shown that this delay is negligible. In addition, we developed two CS policies that enforce fair sharing among CS jobs. We have shown, through our experiments, that one of these policies, Site-EquiPartition, ensures comparable throughput for all CS jobs. An effective CS policy must take into account the heterogeneous and dynamic nature of the grid. Site-EquiPartition does not explicitly consider these properties. However, by putting parts of each job on each available cluster, this policy drastically reduces any negative impact those fundamental grid properties might have.

Our work shows that a CS system must process a relatively large amount of information regarding the grid to perform unobtrusively. A CS system needs to keep track of local queues and grid queues, as well as the tasks that belong to the system's active job. Obviously, a CS system that tracks all this information centrally would allow for more efficient growing and shrinking. However, such a centralized system would suffer from a lack of scalability. Therefore, we believe that the tiered approach of KOALA-CS is preferable, with a KCM on the head node of each cluster and a launcher on each node, that only sends information to the runner when it is necessary, restricting the number of messages sent and the number of programs and scripts remotely executed.

As future work, we would consider the unobtrusiveness for local jobs. First,

the frequency with which the KCM checks the local scheduler's queue should be investigated. We believe that a higher frequency will make KOALA-CS more unobtrusive to local jobs. However, the cost of this is that the KCM will require more processing power. In addition, the KCM may send shrink messages to the runner before the runner could handle previous messages, causing the runner to preempt more launchers than necessary. To make KOALA-CS less unobtrusive to local jobs, therefore, requires careful tuning of both the KCM and the runner.

There is also room for improving the efficiency of local job shrinking. In the current implementation, the shrinking because of local jobs is very drastic and may cause far more launchers to be preempted than strictly necessary. In our current design, each active runner releases the number of nodes needed by the waiting job. However, with each runner having multiple launchers on the same site, the number of nodes to be released should be divided amongst the different runners. While all the information to do this is already available in each runner, the algorithm that governs this local shrinking must not interfere with the CS policy. One way to do this is to relay the information to the scheduler, which must then calculate how many nodes each runner must release. However this will cause timing and synchronization issues.

In this thesis, we have not investigated the effect of Application-Level Scheduling (ALS) policies other than Single task Launcher Pull (SLP). We believe job throughput may be improved by using, for instance, task replication. For data-heavy applications, such as data mining, there could be an ALS policy that takes file locations into account and attempts to minimize transfer times.

More advanced issues include the incorporation of checkpointing, a framework for developing more efficient applications specifically for KOALA-CS, and using KOALA-CS for other types of applications, such as workflows or parallel jobs.

Bibliography

[1] Compute Against Cancer. http://www.computeagainstcancer.org.

[2] Distributed Resource Management Application API. http://www.drmaa.net/w/.

[3] Folding@home. http://folding.stanford.edu/.

[4] Fura grid middleware. http://www.gridsystems.org.

[5] Globus monitoring and discovery system. http://www.globus.org/mds/.

[6] pickle - Python Object Serialization http://docs.python.org/library/pickle.html.

[7] Rosetta@home. http://boinc.bakerlab.org/rosetta/.

[8] SETI@home. http://setiathome.ssl.berkeley.edu/.

[9] The Distributed ASCI Supercomputer 3. http://www.cs.vu.nl/das3/.

[10] The Eternity Puzzle. http://www.eternityii.com.

[11] The Globus Resource Specification Language (RSL) v1.0 http://www.globus.org/toolkit/docs/2.4/gram/rsl_spec1.html.

[12] The Great Internet Mersenne Prime Search (GIMPS). http://www.mersenne.org.

[13] The StarPlane Project. http://www.starplane.org.

[14] The Sun Grid Engine (SGE). http://http://www.sun.com/software/gridware/.

[15] D. Anderson. BOINC: A System for Public-Resource Computing and Storage. In *Proc. of the Fifth IEEE/ACM International Workshop on Grid Computing (GRID'04)*, pages 4–10, 2004.

[16] H. Casanova, A. Legrand, D. Zagorodnov, and F. Berman. Heuristics for Scheduling Parameter Sweep Applications in Grid Environments. In *Proc. of the 9th Heterogeneous Computing Workshop (HCW 2000)*, pages 349–363, 2000.

[17] A. Chien, B. Calder, S. Elbert, and K. Bhatia. Entropia architecture and performance of an enterprise desktop grid system. *Journal of Parallel and Distributed Computing*, 63(5):597–610, 2001.

[18] W. Cirne, F. Brasileiro nd N. Andrade, L. Costa, A. Andrade, R. Novaes, and M. Mowbray. Labs of the world, unite!!! *Journal of Grid Computing*, 3(4):225–246, 2006.

[19] D. Paranhos da Silva, W. Cirne, and F. Vilar Brasileiro. Trading Cycles for Information: Using Replication to Schedule Bag-of-Tasks Applications on Computational Grids. *Lecture Notes in Computer Science*, 2790:169–180, 2004.

[20] D. Epema, M. Livny, R. van Dantzig, X. Evers, and J. Pruyne. A worldwide flock of Condors: load sharing among workstation clusters. *Future Generation Computer Systems*, 12(1):53–65, 1996.

[21] D. Feitelson and A. Weil. Utilization and Predictability in Scheduling the IBM SP2 with Backfilling. *Parallel Processing Symposium, International*, 0:542, 1998.

[22] J. Frey, T.Tannenbaum, I. Foster, M. Livny, and S Tuecke. Condor-G: A Computation Management Agent for Multi-Institutional Grids. *Cluster Computing*, 20:237–246, 2002.

[23] A. Iosup, C. Dumitrescu, D. Epema, H. Li, and L. Wolters. How are Real Grids Used? The Analysis of Four Grid Traces and Its Implications. In *Proc. of the Seventh IEEE/ACM International Conference on Grid Computing (Grid)*, pages 262–269, 2006.

[24] A. Iosup and D. Epema. GRENCHMARK: A Framework for Analyzing, Testing, and Comparing Grids. In *Proc. of the sixth IEEE/ACM International Symposium on Cluster Computing and the GRID (CCGrid'06)*, pages 313–320, 2006.

[25] R. Koo and S. Toueg. Checkpointing and Rollback-Recovery for Distributed Systems. *IEEE Transactions on Software Engineering*, 13(1):1150–1158, 1987.

[26] Y. Lee and A. Zomaya. A Grid Scheduling Algorithm for Bag-of-Tasks Applications Using Multiple Queues with Duplication. In *Proc. of the 5th IEEE/ACIS International Conference on Computer and Information Science and 1st IEEE/ACIS International Workshop on Component-Based Software Engineering,Software Architecture and Reuse (ICIS-COMSAR '06)*, pages 5–10, 2006.

[27] M. Litzkow, M. Livny, and M. Mutka. Condor: A Hunter of Idle Workstations. In *Proc. of the Eigth International Conference on Distributed Computing Systems (ICDCS'88)*, pages 104–111, 1988.

[28] C. McCann and J. Zahorjan. Processor Allocation Policies for Message-Passing Parallel Computers. In *Proc. of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'94)*, pages 19 –32, 1994.

[29] H. Mohamed. *The Design and Implementation of the KOALA Grid Resource Management System*. PhD thesis, Delft University of Technology, 2007.

[30] H. Mohamed and D. Epema. The Design and Implementation of the KOALA Co-allocating Grid Scheduler. *Lecture Notes in Computer Science*, 3470:640–650, 2005.

[31] H. Mohamed and D. Epema. KOALA: A Co-Allocating Grid Scheduler. *Concurrency and Computation: Practice and Experience*, 20:1851–1876, 2008.

[32] M. Mutka and M. Livny. Scheduling remote processing capacity in a workstation-processing bank computing system. In *Proc. of the Seventh International Conference on Distributed Computing Systems*, pages 2–9, 1987.

[33] M. Mutka and M. Livny. The available capacity of a privately owned workstation environment. *Performance Evaluation*, 12(4):269–284, 1991.

[34] I. Raicu, Y. Zhao, C. Dumitrescu, I. Foster, and M. Wilde. Falkon: a fast and light-weight task execution framework. In *Proc. of the 20th International Conference for High Performance Computing, Networking, Storage and Analysis (SC'07)*, pages 1–12, 2007.

[35] O. Sonmez, B. Grundeken, H. Mohamed, A. Iosup, and D. Epema. Scheduling Strategies for Cycle Scavenging in Multicluster Grid Systems. 2008. Accepted by the Ninth IEEE International Symposium on Cluster Computing and the Grid (CCGrid'09).

[36] M. Teodoro, G. Phillips, and L. Kavraki. Molecular Docking: A Problem with Thousands of Degrees of Freedom. In *Proc. of the 2001 IEEE International Conference on Robotics and Automation (ICRA'01)*, pages 960–966, 2001.