# The Design and Implementation of the KOALA Grid Resource Management System

Hashim H. Mohamed

# The Design and Implementation of the KOALA Grid Resource Management System

**Proefschrift**

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus prof.dr.ir. J.T. Fokkema,
voorzitter van het College voor Promoties,
in het openbaar te verdedigen op maandag 26 november 2007 om 15:00 uur

door **Hashim Hussein MOHAMED**

informatica ingenieur
geboren te Arusha, Tanzania

Dit proefschrift is goedgekeurd door de promotor:
Prof.dr.ir. H.J. Sips

*Samenstelling promotiecommissie:*

| | |
|---|---|
| Rector Magnificus | voorzitter |
| Prof.dr.ir. H.J. Sips | Technische Universiteit Delft, promotor |
| Dr.ir. D.H.J. Epema | Technische Universiteit Delft, toegevoegd promotor |
| Prof.dr.ir. H.E. Bal | Vrije Universiteit Amsterdam |
| Prof.dr. F. Desprez | INRIA, France |
| Prof.dr.ir. A.J.C. van Gemund | Technische Universiteit Delft |
| Prof.dr. C. Kesselman | University of Southern California, USA |
| Dr. A.A. Wolters | Universiteit Leiden |
| Prof.dr. C. Witteveen | Technische Universiteit Delft, reservelid |

Published and distributed by: Hashim H. Mohamed
E-mail: hashim@hashspace.net

ISBN: 978-90-9022579-1

Keywords: Grids, clusters, scheduling, co-allocation, deployment, experimentation, performance



Printed in The Netherlands by: PrintPartners Ipskamp

*To my wife Mgeni,*
*with lots of love.*

# Contents

# Chapter 1

# Introduction

Grid computing has emerged as an important new field in computer systems, distinguished from conventional distributed computing by its focus on large-scale, multi-organizational resource sharing and innovative applications. At the heart of grid computing is a computing infrastructure that provides ubiquitous and inexpensive access to large amounts of computational capabilities [65]. Over the past 15 years, we have seen a substantial growth of the grid hardware and software infrastructure. The hardware growth is mainly due to the increase in the performance of commodity computers and networks, which has been accompanied by a drop in their prices. On the other hand, grid software for building single-cluster and multicluster systems and grid middleware technologies have become more sophisticated and robust. Multicluster systems are formed by joining multiple, geographically distributed clusters interconnected by high-speed wide-area networks. An example of a multicluster system is the Distributed ASCI Supercomputer (DAS), which will be discussed in detail in Section 2.1.1. Grid middleware sits between grid applications and the grid hardware infrastructure and therefore, hides the underlying physical infrastructure from the users and from the vast majority of programmers. In doing so, grid middleware offers transparent access to a wide variety of distributed resources to users and simplifies the collaboration between organizations. As a result of the growth of the grid infrastructure and what it promises to offer, new applications and application technologies have emerged that are attempting to take advantage of the grid. These applications have widely different characteristics that pose unique resource requirements to the grid.

Resource management is an important subject in multiclusters and grids, and the software implementing the mechanisms and policies for resource management constitutes a large fraction of the grid middleware. Although grids usually consist of many subsystems such as clusters and multiprocessors, and jobs submitted to grids might benefit from using resources in multiple such subsystems, most jobs in grids use the resources of only one such subsystem. However, some types of jobs, e.g., jobs that run parallel applica-

tions that can efficiently use many processors, may take advantage from using resources in multiple subsystems of a grid. Therefore, in multicluster systems, and more generally, in grids, jobs may require *co-allocation*, i.e., the simultaneous or coordinated access of single applications to resources of possibly multiple types in multiple locations managed by different autonomous resource management systems [31, 51, 78]. Co-allocation presents new challenges to resource management in grids, such as coordinating the times of access of a job to resources in different clusters. These challenges, which we address in this thesis, are presented in Section 1.1. The fact that grid applications have unique resource needs and have unique ways of being deployed in the grid, forms another challenge that we also address in this thesis. This challenge is described in Section 1.2. In Section 1.3 we give an overview of our approach of meeting all of these challenges by means of the design and the implementation of the KOALA Grid Resource Management System. In Sections 1.4 and 1.5, we state the contributions of this thesis to the research in grid resource management, and we present an overview of the whole thesis.

## 1.1 Challenges in Resource Management in Grids

Grids need high-level schedulers that can be used to manage resources across multiple organizations. Such schedulers have variably been called (albeit with somewhat different meanings) resource brokers, meta-schedulers, higher-level schedulers, superschedulers, grid schedulers, grid resource management systems (GRMS), etc. In this thesis we will stick to the latter term. GRMSs have important characteristics that make them much more complicated to design and implement than Local Resource Management Systems (LRMS) for single clusters. These characteristics, which lead to challenges in resource management in grids, are:

1. GRMSs do not own resources themselves, and therefore do not have control over them; they have to interface to information services about resource availability, and to LRMSs to schedule jobs. Grids are usually collections of clusters (or of other types of computer systems such as multiprocessors and supercomputers) that have different owners, that have their own user community, and that have their own, autonomous local scheduler. These owners are often not willing to give up the autonomy of their clusters, but will only allow access to their resources through a GRMS that interfaces to their local schedulers according to specific usage rules.

2. GRMSs do not have a full control over the entire set of jobs in a grid; local jobs and jobs submitted by multiple GRMSs have to co-exist in a grid. The jobs that are executed in a single cluster in a grid may be submitted through the local scheduler or through any of a number of GRMSs. This means that a GRMS has to take into

account jobs from multiple sources when deciding on where a particular job should run.

3. GRMSs have to interface to different LRMSs with different properties and capabilities. At the time of writing of this thesis, the ongoing standardization effort of the interface between GRMSs and LRMSs by the OGF [19] was far from complete.

An important possible requirement to a GRMS is to support co-allocation. The problem of co-allocation and of adding support for it to a GRMS is at the core of this thesis. In Section 1.1.1 we elaborate on the co-allocation problem that we address. Co-allocation relies on the simultaneous availability of resources, which is simplified by the presence of mechanisms for advance resource reservations, in particular for processors, in LRMSs. In Section 1.1.2 we discuss issues regarding advance processor reservations in LRMSs.

### 1.1.1 The co-allocation problem

In grids, it is common for the resource needs of grid applications to go beyond what is available in any of the sites making up a grid. For example, a parallel application may require more processors than are present at any site, and a simulation may require processors for computation in one site and visualization equipment in another site. To run such applications, co-allocation, defined as the simultaneous or coordinated access to resources of possibly multiple types in multiple locations managed by different resource management systems, is required. When co-allocation is employed for a job, we call the parts of the job that run at different sites *job components*. Co-allocation presents the following challenges in resource management in grids:

1. *Allocating resources in multiple sites, which may be heterogeneous in terms of hardware, software, and access and usage policies, to single applications.* When co-allocating resources, the primary goal of a co-allocation mechanism is to minimize the waiting time as well as the response time of jobs. In order to do so, the hardware type and speed (e.g., processor speed, network bandwidth), the presence of the required software (e.g., the operating system, libraries), and the usage policies need to be taken into account when co-allocating resources to jobs.

2. *Guaranteeing the simultaneous availability of the co-allocated resources at the start of the execution of an application.* A co-allocation mechanism is only effective if after co-allocating resources, those resources can actually be used. The facts that GRMSs do not own resources, that grid resources are very dynamic, and that there is contention for resources between local and global grid jobs make guaranteeing resource availability to jobs a real problem. This problem is elaborated on more in Section 1.1.2.

3. *Managing sets of highly dynamic grid resources belonging to multiple sites of a grid, which may come and go at any time, either by being disconnected or by failing.* Grids harness the power of many networked clusters, desktop computers, and even scientific instruments from different organizations. The levels of resource availability between organizations differ; some claim 5-nines availability while others have high failure rates. When co-allocating resources, it is important to consider their reliability, and in case of failures, good error recovery mechanisms should be present in GRMSs.

### 1.1.2 Processor reservations

The challenge with simultaneous access of an application to resources at multiple sites of a grid lies in guaranteeing their availability at the application's start time. The most straightforward strategy to do so is to reserve processors at each of the selected sites. If the LMRSs do support reservations, this strategy can be implemented by having a GRMS obtain a list of available time slots from each LRMS, reserve a common time slot for all job components, and notify the LRMSs of this reservation. Unfortunately, a reservation-based strategy in grids is currently limited due to the fact that only few LRMSs support reservations (for instance, PBS-pro [23] and Maui [15] do). Even for those resource managers, only privileged users or specially designated user groups are allowed to perform processor reservations, in order to prevent users from abusing the reservation mechanism. In the absence of, or in the presence of only limited processor-reservation mechanisms, good alternatives are required in order to achieve co-allocation.

## 1.2 The Challenge of Deploying Grid Applications

While grid infrastructures have become almost common-place, the automation of grid resource management is far from complete because of the complexity of the applications that occur in practice [55]. These applications bear different characteristics and pose unique requirements to GRMSs. In addition to the application characteristics, the characteristics of the grid infrastructure itself complicate the deployment of jobs by grid resource management tools. In Sections 1.2.1 and 1.2.2 we discuss characteristics of grid applications and of grid infrastructures that make the deployment of jobs in grids a challenge.

### 1.2.1 Application characteristics

Over the last decade the number of grid applications has increased considerably, causing a new challenge in automating their deployment on grids. This challenge stems from the special characteristics of the applications that pose unique requirements to job deployment

mechanisms. The following application characteristics complicate the automation of the deployment of jobs on grids:

1. *The application structure in terms of the number of job components and their resource requirements.* The application structure determines the number of components and the resource requirements of each of the components. For example, *rigid jobs* require fixed numbers of components and of processors, which do not change during the execution of the application. On the other hand, a *malleable job* requires a flexible numbers of components and of processors, which may change during the execution of the application. Moreover, a parallel application, whether it is rigid or malleable, may require more processors than are available at any site, and a simulation may require processors for computation in one site and visualization equipment in another site. To run such applications, a deployment mechanism has to be co-allocation-enabled, and in case of malleable jobs, it also has to be malleable-enabled.

2. *The communication structure within and between job components*. The communication structure within and between the components of a job is vital to the successful execution of grid applications. It requires a job deployment mechanism to assist in the setup of the communication between the job components, and to link the application to the correct communication library. For example, the communication structure of an application may require a centralized server to set up communication between its components and coordinate its execution. For instance, Ibis applications need a nameserver as described in Section 2.1.8. The deployment mechanism for Ibis jobs is required to start this centralized server before launching jobs for execution, and to make jobs belonging to the same computation aware of the same centralized server.

3. *The need for a particular runtime system.* Each grid application type has its own runtime system that has to be present during the execution of the application, possibly in multiple sites. A deployment mechanism is required to ensure that the correct runtime system is present at the sites in question before launching jobs for execution.

### 1.2.2 Grid infrastructure characteristics

The grid infrastructure forms the foundation for the successful deployment of applications on grids. This infrastructure, which spans multiple organizations, consists of different hardware and software providing capabilities and resources for the specific problems being addressed by each organization. The following grid infrastructure characteristics,

some of which in fact were also at the basis of the challenges in co-allocation as stated in Section 1.1.1, complicate the automation of the deployment of jobs on grids:

1. *The grid infrastructure is highly heterogeneous in terms of the grid software in use in different sites.* The grid software includes grid middleware software such as the Globus Toolkit [22, 63], UNICORE [92], DIET [4] and gLite [12], and LRMSs such as openPBS [24] and PBSPro [23], SGE [26], and Condor [95]. Different grid software have different properties and capabilities that complicate the deployment of jobs. For example, not all LRMSs support automatic advance processor reservations [78]. Therefore, jobs that are spawned across multiple domains often require users to coordinate resource availability by hand. A job deployment mechanism is required to cope with the heterogeneity of the grid and to simplify the job submission procedure for users.

2. *Grid resources are highly dynamic.* A deployment mechanism is required to have good fault tolerance mechanisms to ensure that jobs are executed successfully despite the dynamicity of the grid.

3. *Grid resources have to be configured for each application, in particular with respect to the network and security*. Firewalls, hidden/private IP addresses and Network Address Translation (NAT) hamper connectivity, while authentication and encryption mechanisms in different grid domains are usually difficult to integrate. Again, a deployment mechanism is required to hide the configuration issues of the grid from the users without tampering with the authentication mechanisms and network configurations.

## 1.3 An Approach to Resource Management and Job Deployment on Grids

In order to address the challenges of co-allocation and of grid application deployment, we have designed, implemented, and deployed in the DAS system (see Section 2.1.1) a GRMS called KOALA, which features co-allocation of processors and files; the name KOALA was solely chosen for its similarity in sound with the word co-allocation. In addition, in order to assess the operation and some performance aspects of KOALA, we have performed experiments with KOALA in the DAS.

To meet the challenge of allocating resources in multiple sites, KOALA has built-in so-called placement policies for allocating processors in multiple clusters to single jobs. Currently, there are two such policies: the Close-to-Files (CF) policy and the Worst Fit (WF) policy. New placement policies can be added without affecting the overall operation

of KOALA. CF addresses the problem of long delays when starting a job because of long input file transfers by selecting the execution sites of its components close to sites where their input files are located. On the other hand, the WF policy simply places job components on the sites with the largest numbers of idle processors. In doing so, WF balances the numbers of idle processors across the grid. The placement policies extensively use the KOALA Information Service to locate and monitor resource availability. Users have a choice of which placement policy to use for every job they submit separately.

Due to potentially long input file transfer times, the actual start of a job's execution (its job start time) may be much later than the time when the job was allocated processors (its job placement time). This means that when the allocated processors are claimed immediately at the job placement time, much processor time is wasted. In order to prevent this and meet the challenge of guaranteeing the simultaneous availability of processors at the job start time in the absence of support for advance processor reservation by LRMSs, KOALA implements the Incremental Claiming Policy (ICP). If needed because some of the allocated processors have been taken by other jobs, in an effort not to delay the job start time, ICP tries to make processors available for job components by finding processors at other sites or, if permitted, by forcing processor availability through preemption of running jobs.

To address the challenge of grid application deployment, KOALA introduces the concept of *runners*, which are job submission and monitoring tools. Different runners can be written to support the unique characteristics of different application types by using the KOALA runners framework. The runners framework within KOALA is modular and allows new runners for new application types to be added without affecting the current operation of the existing runners. For example, new runners have been written specifically for Higher-Order Component applications [55] and for malleable applications [42] by novices to KOALA with minimal effort. The runners framework has fault tolerance mechanisms that deal with the reliability issues of the grid infrastructure. The part of KOALA that performs scheduling of jobs and the runners framework work together to meet the challenge of managing sets of highly dynamic grid resources.

An important aspect of any component of grid middleware, and even more so of a scheduler, is its performance. Therefore, in order to assess the reliable operation and some of the performance properties of KOALA, we have performed experiments in which we submit workloads consisting of parallel jobs that require processor and data co-allocation to the DAS through KOALA. For instance, we submit workloads imposing different utilizations on the DAS to assess to what utilization we can drive the DAS when employing co-allocation, to assess the response times of co-allocated jobs distinguished by their numbers of job components and job-component sizes, and to assess the overhead incurred when starting jobs through KOALA. KOALA has a number of parameters that can be tuned in a particular installation. This thesis does not include a full parameter study of the

operation of KOALA, nor does it present a full performance analysis of co-allocation in multicluster systems.

## 1.4 Contributions of this Thesis

The major contributions of this thesis are the following:

1. The design, the implementation, and the deployment of a reliable co-allocating grid scheduler called KOALA, and the demonstration of its correct operation in the DAS testbed.

2. The design and the analysis of two co-allocation policies, the Close-to-Files policy, which takes into account the locations of input files in addition to the availability of processors in the clusters, and the Worst-Fit policy, which balances jobs across the clusters.

3. The design and the analysis of a processor claiming policy called the Incremental Claiming Policy as an alternative to advance processor reservation when such a mechanism is absent in the LRMSs for achieving the simultaneous availability of allocated processors.

4. The design, the implementation, and the deployment of the runners framework and of three runners for deploying different grid application types, and the demonstration of their correct operation in the DAS testbed.

## 1.5 Thesis Outline

The material in this thesis is structured as follows:

- In Chapter 2 we give an overview of resource management in grids that is necessary for reading this thesis. We present our model for resource management in grids, and we discuss grid applications. In addition, we review related work.

- In Chapter 3 we present the design of our KOALA grid resource management system. The architecture of KOALA can be divided into two major layers, namely, the KOALA scheduler and the runners, which are job submission and monitoring tools.

- In Chapter 4 the KOALA job policies that are used to schedule jobs and to claim processors are presented.

- In Chapter 5 we present the evaluation of the KOALA scheduler, which includes an evaluation of the job policies and of the scheduler in general, both on a stable and an unstable testbed.
- In Chapter 6 we evaluate the KOALA runners. The experiments presented in this chapter evaluate both the functionality as well as the performance of the runners.
- Chapter 7 presents our conclusions and some open research questions.

## 1.6 List of Publications

Below is the list of publications of the author of this thesis.

1. H.H. Mohamed and D.H.J Epema. Interfacing Different Application Types to the KOALA Grid Scheduler. Submitted for publication.
2. H.H. Mohamed and D.H.J Epema. KOALA: A Co-Allocating Grid Scheduler. *Concurrency and Computation: Practice and Experience*. Accepted for publication.
3. C. Dumitrescu, A. Iosup, O. Sonmez, H.H. Mohamed, and D.H.J. Epema. Virtual Domain Sharing in e-Science Based on Usage Service Level Agreements. In *Proc. of the CoreGRID Symposium*, Rennes, France, CoreGRID series, Springer-Verlag, pages 15–26, 2007.
4. J. Buisson, H.H. Mohamed, O. Sonmez, W. Lammers, and D.H.J. Epema. Scheduling Malleable Applications in Multicluster Systems. In *Proc. of the IEEE International Conference on Cluster Computing 2007*, pages 372–381, 2007.
5. O. Sonmez, H.H. Mohamed, and D.H.J. Epema. Communication-Aware Job Placement Policies for the KOALA Grid Scheduler. In *Proc. of the Second IEEE International Conference on e-Science and Grid Computing*, pages 79–87 , 2006.
6. A.I.D. Bucur, H.H. Mohamed, and D.H.J. Epema, Co-allocation in Grids: Experiences and Problems. In *Proc. of the Workshop on Future Generation Grids*, Dagstuhl, Germany, CoreGRID series, Springer-Verlag, pages 195–213, 2006.
7. H.H. Mohamed and D.H.J. Epema. The Design and Implementation of the KOALA Co-Allocating Grid Scheduler. In *European Grid Conference*, volume 3470 of LNCS, pages 640–650, 2005.
8. H.H. Mohamed and D.H.J. Epema. An Evaluation of the Close-to-Files Processor and Data Co-Allocation Policy in Multiclusters. In *Proc. of the IEEE International Conference on Cluster Computing 2004*, pages 287–298, 2004.

9. J.M.P. Sinaga, H.H. Mohamed, and D.H.J. Epema. A Dynamic Co-Allocation Service in Multicluster Systems. In *Proc. of the 10th Workshop on Job Scheduling Strategies for Parallel Processing*, volume 3277 of LNCS, pages 194–209, 2004.

10. H.H. Mohamed and D.H.J. Epema. A Prototype for Processor and Data Co-Allocation in Multiclusters. In *Proc. of the ASCI 2004 Conference*, pages 243–250, 2004.

# Chapter 2

# Resource Management in Grids

Grids have a goal of offering transparent access to large collections of resources for applications demanding many processors and access to huge data sets. To realize this goal, resource management in grids is crucial. We begin this chapter in Section 2.1 by giving the background on grids required to read our work. In Section 2.2, we present the model for resource management in grids that is used in this thesis. Grid applications, including the sample grid applications that are used in our experiments in later chapters, are introduced in Section 2.3. Finally, in Section 2.4 we review related work.

## 2.1 Background

This section presents the background that is required to read this thesis. This background includes the detailed description of a multicluster system in Section 2.1.1 called the Distributed ASCI Supercomputer (DAS), which was an important motivation for our work. We then describe the software infrastructure of the grid, namely, local resource management systems, grid middleware, grid resource management systems, and grid programming models in Sections 2.1.2–2.1.8.

### 2.1.1 The DAS system

The Distributed ASCI Supercomputer (DAS) [21] is an experimental computer testbed in the Netherlands that is exclusively used for research on parallel, distributed, and grid computing. This research includes work on the efficient execution of parallel applications in wide-area systems [103, 104], on communication libraries optimized for wide-area use [17, 98], on programming environments [98, 99], and on resource management


Figure 2.1: The five clusters of the Distributed ASCI Supercomputer 3.

and scheduling [34, 39, 40]. The system was built for the Advanced School for Computing and Imaging (ASCI), a Dutch research school in which several universities participate. The DAS is now entering its third generation, after the first and second generations have proven to be successes.

The first generation DAS system (DAS-1) [35, 61] consisted of four clusters of identical Pentium Pro processors, one cluster with 128 processors and three with 24 processors. The clusters were interconnected by ATM links for wide-area communications, while for local-area communications within the clusters, Myrinet LANs were used. On single DAS clusters a local scheduler called *prun* was used that allowed users to request a number of processors bounded by the clusters' sizes.

The first generation was replaced by the second generation of the DAS system (DAS-2) at the end of 2001. The DAS-2 consisted of 200 nodes, organized into five dual-CPU clusters as shown in Table 2.1 of identical 1GHZ Intel Pentium III processors. For local communication within the single clusters, low-latency Myrinet-2000 LAN was used. The clusters were interconnected by SURFnet5, the Dutch education and research gigabit backbone. Until mid 2005, all the DAS-2 clusters used openPBS [24] as the local resource manager. However, due to reliability problems after the latest upgrade of openPBS, the decision was made to change the local resource manager of the clusters to the Sun N1 Grid Engine (SGE) [26].

The DAS system is now in its third generation (DAS-3), which was installed late 2006 (see Figure 2.1). In some of the universities where the DAS clusters are hosted, the decision was made to still maintain the DAS-2 clusters, thus making the DAS system

Table 2.1: The distribution of the nodes over the DAS-2 clusters.

| Cluster Location | Number of Nodes |
|---|---|
| Vrije University | 72 |
| Leiden University | 32 |
| University of Amsterdam | 32 |
| Delft University | 32 |
| Utrecht University | 32 |

more heterogeneous. The DAS-3 consists of 272 nodes organized into five dual-CPU clusters as shown in Table 2.2 with a mixture of single-core and dual-core AMD Opteron processors. All the DAS-3 clusters have 1 Gb/s and 10 Gb/s Ethernet, as well as high speed Myri-10G [18] interconnect, except the cluster in Delft, which has only Ethernet interconnects. For wide-area communications, initially 1Gb/s connectivity provided by SURFnet6 was used, but at the time of writing this thesis, there is an ongoing collaborative effort between the StarPlane project [25] and SURFnet to enable DAS-3 to use dedicated 10Gb/s lightpaths between clusters.

Table 2.2: The distribution of the nodes over the DAS-3 clusters.

| Cluster Location | Number of Nodes |
|---|---|
| Vrije University | 85 |
| Leiden University | 32 |
| University of Amsterdam | 41 |
| Delft University | 68 |
| The MultimediaN Consortium | 46 |

In the DAS systems, each of the DAS cluster is an autonomous system with its own file system. Therefore, in principle files (including executables) have to be moved explicitly between users' working spaces in different clusters. Simple usage rules are enforced in the DAS. The most important of these are that any application cannot run for more than 15 minutes from 08:00 to 20:00, and that application execution must be performed on the compute nodes. The DAS systems can be seen as a fast prototyping computational grid environment, with its structure and usage policies designed to make grid research easier.

Other systems such as Grid'5000 [7] and the Open Science Grid (OSG) [62] are similar to the DAS. The Grid'5000 project aims at building a highly reconfigurable, controllable and monitorable experimental grid platform distributed across nine geographically distributed sites in France and is intended to feature a total of 5000 CPUs [7, 44]. There is an ongoing effort of joining the DAS-3 and Grid'5000 [43].

OSG, previously known as Grid3 [62], is a multi-virtual organization environment that sustains production level services required by various physics experiments. The infras-

tructure comprises more than 50 sites and 4500 CPUs, and serves over 1300 simultaneous jobs and more than 2 TB/day aggregate data traffic [56].

### 2.1.2 Local resource management systems

Single clusters, whether they are part of a grid or not, are managed by Local Resource Management Systems (LRMSs), which provide an interface for user-submitted jobs to be allocated resources and to be executed on the cluster. LRMSs support the following four main functionalities: resource management, job queueing, job scheduling, and job execution [107]. Jobs submitted to a cluster are initially placed into queues until there are available resources to execute the jobs. After that, the LRMS dispatches the jobs to the assigned nodes and manages the job execution before returning the results to the users [107]. Most LRMSs such as Condor [3, 95], the Portable Batch System (PBS) [24], and the Sun Grid Engine (SGE) [26] focus on maximizing processor throughput and utilization, and minimizing the average wait time and response time of the jobs.

The LRMSs are typically designed for single administrative domains, and therefore, employ a limited set of policies that tend to favor local jobs. This means that LRMSs do not provide a complete solution to grid resource management problems, although their resource management solutions are an important part of a global grid resource management architecture.

### 2.1.3 Grid middleware

The grid middleware sits between grid applications and the physical resources and therefore, it hides the underlying infrastructure from the users and from the vast majority of programmers. In doing so, the middleware offers transparent access to a wide variety of distributed resources to users and allows the development of collaborative efforts between organizations.

Grid middleware such as the Globus Toolkit [22, 63], Legion [67], UNICORE [92], DIET [4, 46] and gLite [12] have contributed a lot to the growth of grids. Of these grid middlewares, the Globus Toolkit, which is also used in the DAS, is the best known. Section 2.1.4 presents key features of the Globus Toolkit and in Section 2.1.5 we discuss Grid Resource Management Systems (GRMSs), which are built on top of the grid middlewares.

### 2.1.4 The Globus Toolkit

The Globus Toolkit comprises a set of modules, each of which defines an interface that higher-level services use to invoke that module's mechanisms. The Globus Toolkit uses appropriate low-level operations to implement these mechanisms in different environments [63]. The modules provided with the Globus Toolkit are for resource location and

allocation, communications, unified resource information service, authentication, process creation, and data access. We will discuss two of these modules, those for resource location and allocation and for authentication, which are important to our work.

The resource location and allocation module provides a mechanism for expressing application resource requirements in the Globus Resource Specification Language (RSL) [22], and for scheduling resources once they have been located through the Globus Resource Allocation Manager (GRAM) [22]. We should point out that the most common usage of GRAM is not scheduling resources by itself, but rather mapping the resource specification onto a request of some LRMS such as PBS, SGE, Condor, Fork, or LSF, which in turn does the scheduling on a remote site. This allows GRAM to inter-operate with autonomous heterogenous sites that use different LRMSs. As a result of interfacing with different LRMSs, GRAM provides an ideal interface between GRMSs and autonomous remote sites (see Figure 2.2). It should be noted here that the process of locating resources is left to GRMSs to accomplish, which is the subject of the next section.

The authentication provided by the Globus Security Infrastructure (GSI) [22] is the core module of the Globus Toolkit, which provides basic authentication mechanisms that can be used to validate the identity of both users and resources. A central concept in GSI authentication is the certificate by which every user or service on grids is identified. The grid certificate contains vital information necessary for identifying and authenticating the user or service. The GSI supports delegation of credentials for computations that involve multiple resources and/or sites. This allows a user to sign-on only once (single sign-on) to use grid resources in multiple sites.

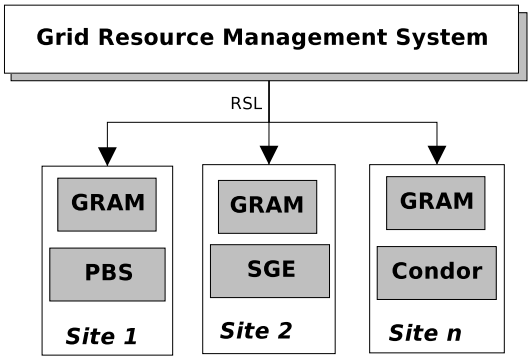



Figure 2.2: The GRAM providing an interface to different local resource management systems.

### 2.1.5 Grid resource management systems

Grids need high-level resource management systems built on top of grid middlewares that can schedule (and co-allocate) jobs across multiple sites. The grid resource management system provides an interface to users for submitting grid jobs, for scheduling jobs across the grid, for launching jobs for execution, for error handling, and for recovery during the execution of the job. In addition to scheduling jobs, a well-designed GRMS can provide an environment to perform application level scheduling.

To schedule jobs across the grid, good scheduling algorithms are required to identify sufficient sites for the jobs based on the information obtained from the grid resource manager's information provider. The information provider needs to be reliable and must have dynamic and static information about the availability of grid resources. With some grid resource managers, the estimated execution time of a computation as specified by a user, and cost constraints in addition to the information provided by their information providers, are used when scheduling jobs [27]. For grid resource managers managing resources spanning multiple organizations, managing usage service level agreements [50, 56, 57] may be required as well.

A GRMS built on top of different grid middlewares can be thought of as defining a metacomputing virtual machine. The virtual machine simplifies application development and enhances portability by allowing programmers to think of geographically distributed, heterogeneous collections of resources as unified entities.

### 2.1.6 Grid programming models

Grid programming models hide the heterogeneity of the grid and of the resources to grid application programmers. Additionally, an application written with a grid programming model is essentially shielded from future, potentially disruptive changes in the grid middleware. In this thesis, the term "programming model" is used to include both programming languages and libraries with APIs that encode abstractions of commonly used grid operations [71]. Such grid operations include wide-area communication, error handling, adaptivity to resource availability, checkpointing, job migration, distributed event models for interactive applications, and collaborative interfaces. Grid programming models are relevant to grid resource management, as GRMSs have to be able to schedule, launch and monitor applications written using these models. We call grid applications that are written from scratch with execution on a grid in mind using a certain grid programming model *native grid applications*. Examples of grid programming models include *Ibis* [97, 98], MPICH-G2 [17], the Grid Application Toolkit (GAT) [1], and gridRPC [89].

Grid programming models can also be used to grid-enable legacy applications. Grid-enabling these applications is achieved by interfacing the application codes with a suitable grid programming model, which can be thought of as being part of the grid middleware.

We call grid-enabled legacy applications *legacy grid applications*.

The presence of grid programming models has resulted in the creation of different grid applications, posing the new challenge of automating their deployment on different sites by grid resource management tools. This is because the programming model dictates how these applications should be deployed, first to be able to run them successfully, and secondly, to optimally utilize grid resources and therefore, to improve their performance.

### 2.1.7 The grid-enabled Message Passing Interface

The Message Passing Interface (MPI) is a widely known standard that defines a two-sided message passing (matched sends and receives) library that is used for parallel applications and is well suited for grids. Many implementations of MPI exist, amongst which MPICH-G2 [17] is the most prominent for grid computing. It allows the coupling of multiple sites to run a single MPI application by automatically handling both inter-site and intra-site messaging. MPICH-G2 requires Globus services to be available on all participating sites, and therefore, the co-allocation of MPICH-G2 jobs is limited to clusters with the Globus middleware installed.

### 2.1.8 The Ibis grid programming system

Ibis has been developed at the Vrije University in Amsterdam and has as its goal the design and implemention of an efficient and flexible Java-based programming environment and runtime systems for grid computing [97,98]. Currently, Ibis offers four grid programming models: Remote Method Invocation (RMI) [74], which is Java's object-oriented equivalent of RPC, Group Method Invocation (GMI) [74, 75], which extends RMI with group communication, Replicated Method Invocation (RepMI) [74], which extends Java with efficient replicated objects, and Satin [96], which provides the divide-and-conquer and replicated-worker programming models. Key to the design of Ibis is to achieve a system that obtains high communication performance while still adhering to Java's "write once, run anywhere" model. Ibis is designed to use any standard JVM including standard communication protocols, e.g., TCP/IP or UDP, as provided by the JVM. However, if a native optimizing compiler (e.g., Manta [97]) for a target machine and/or optimized low-level protocols for a high-speed interconnect, like GM or MPI, is/are available, then Ibis can use them instead. To run an Ibis job, a central component called the Ibis nameserver is required to coordinate the setup of communication ports between the components of a job running in multiple sites. The presence of the nameserver allows Ibis to support malleable jobs in which nodes participating in a parallel computation can join and leave the run at any time. Also, the nameserver can be used to route messages between clusters if a direct connection is not possible. A single instance of the Ibis nameserver can be instantiated

for each job or can be shared among several Ibis jobs. While starting the nameserver for each job is expensive in terms of resource usage, sharing the same Ibis nameserver can be a bottleneck.

## 2.2 A Model for Resource Management in Grids

This section presents the model for resource management in grids that is used in this thesis. Section 2.2.1 discusses the system model, which is inspired by the DAS system. The job model, including the structure of the job requests and job priorities are described in Sections 2.2.2 and 2.2.3. This section also presents the file distribution model that we use in Section 2.2.2.

### 2.2.1 The system model

In the system model, we assume a multicluster environment like the DAS with sites that each contain computational resources (processors), a head node, and a local resource manager. In our model, head nodes are normally used as file servers and have huge disk space. The storage system providing the disk space can be directly attached to the local head nodes or to a remote storage system, and accessed via the head nodes through a global/network file system. The sites may combine their resources to be scheduled by a *grid scheduler* for executing jobs in a grid. The sites where the components (see Section 2.2.2) of a job run are called its *execution sites*, and the site(s) where its input file(s) reside are its *file sites*. We assume a grid scheduler through which all grid job submissions are made. The sites where the submissions are made from are called the *submission sites*. A submission site can be any site in a grid, or a desktop computer. The grid scheduler allows us to perform resource brokering and scheduling across its authorized *domain* in the grid about which it has global knowledge of the state of the nodes and network.

### 2.2.2 The job model

In this thesis, a job consists of one or more job components which collectively perform a useful task for a user. The job components contain information such as their numbers and speeds of processors, the sizes and locations of their input files, their memory requirements, and the required runtime libraries necessary for scheduling and executing an application across different sites in a grid. Due to the nature of our research, only numbers of processors, locations and sizes of input files, and network bandwidth are used when scheduling job components.

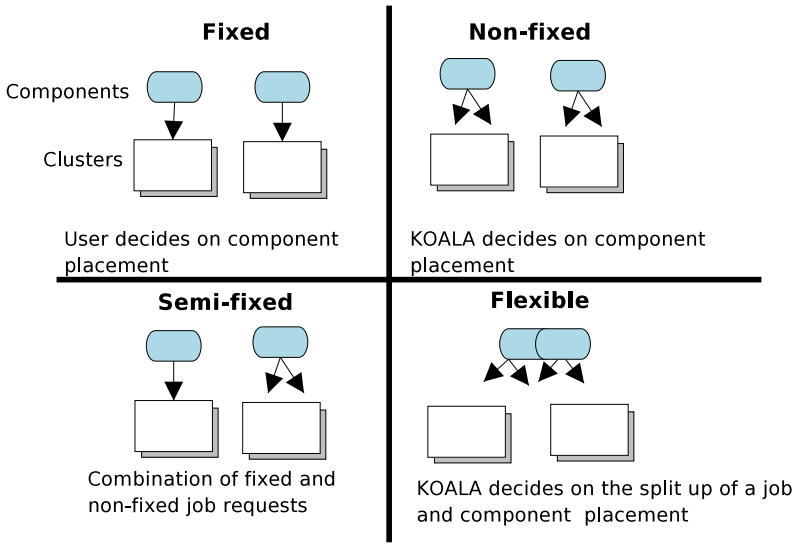



Figure 2.3: The KOALA job request types.

Users need to construct job requests containing a detailed description of the job components as described above. A job request may or may not specify its execution sites and the numbers and sizes (in terms of the number of processors) of its job components. Based on this, we consider four cases for the structure of the job requests, which are depicted in Figure 2.3 and discussed below:

1. **Fixed request**: The job request specifies the numbers of processors it needs in all clusters from which processors must be allocated for its components. With fixed requests, the user decides where the components of the job will run. Fixed requests are useful for jobs that require resources only present at specific sites, such as visualization equipment and software licenses. In general, fixed requests give users the ability to run jobs in more familiar clusters where they know beforehand their performance.

2. **Non-fixed request**: The job request only specifies the numbers of processors required by its components, allowing the scheduler to choose the execution sites. The scheduler can either place the job components on the same or on different sites depending on the availability of processors. Any application type that can run in any cluster should be amenable for submission with a non-fixed job request. For example, the presence of the nameserver in Ibis allows Ibis jobs to run anywhere. MPICH-G2 jobs can also run in any Globus-middleware clusters. Other application types that are computation-intensive like Bags of Tasks (BoTs) and Parameter Sweep Applications (PSAs) are also very well suited to be submitted with non-fixed requests. BoTs and PSAs are discussed in Section 2.3.1.

3. **Semi-fixed request**: The job request is a combination of a fixed and a non-fixed request in that the execution sites for some job components are specified and for others they are not. An example of the use of semi-fixed requests is constituted by applications types that perform simulations in grids and visualization in desktop computers. These application types may require the clusters that interface with the desktop machines to be fixed and the rest that do the computation to be non-fixed.

4. **Flexible request**: The job request only specifies the total number of processors it requires. It is left to the scheduler to split up the job and to decide on the number of components, the number of processors of each component, and the execution sites for the components. With flexible job requests a restriction may be imposed on the number and sizes of the components. For instance, a lower bound or an upper bound may be imposed on the number of components and their sizes. Flexible job requests are not supported by all applications, because some applications such as the Poisson application (see Section 2.3.2) dictate specific patterns of being split up into components. In general, flexible requests are useful for applications that require a large number of processors but do not require a specific pattern of splitting, e.g., BoTs and PSAs.

In the above mentioned job request types, the number of processors required by the job is set by the user and cannot change during the job's execution, i.e., we assume rigid jobs. In all cases, the scheduler has the task of moving the executables as well as the input files to the execution sites before the job starts, and to start the job components simultaneously if required. This is because no matter what the application, it generally requires input data and will produce output data. One of the things we need to consider here is the management of the input data and the gathering of the output data.

In our model, we deal with two models of data distribution to the job components. In the first model, the job components work on different chunks of the same data file, which has been partitioned as requested by the components. This is useful when the data file is large and the job components have been placed on geographically distributed sites. Note this model includes the model in which all job components have different input files that are all stored on the same file sites. In the second model, which is useful for small input files, the input to each of the job components is the whole data file. In both models, data need to be transferred to the execution sites before the job execution starts such that the job can access the data locally. We assume that the data files are read-only, and therefore, that they can be shared by other jobs. This is a reasonable assumption as discussed in several Data Grid scenarios [80].

The input data files have unique logical names and are stored and possibly replicated at different file sites. A logical name of a file is a unique identifier for the file. We assume that there is a replica manager that maps the logical file names specified by jobs onto their

physical file names on a storage system. Figure 2.4 shows an example of a user requesting a physical name of a file. The replica manager replies with the mapping of the file to the physical location(s) on the storage systems.

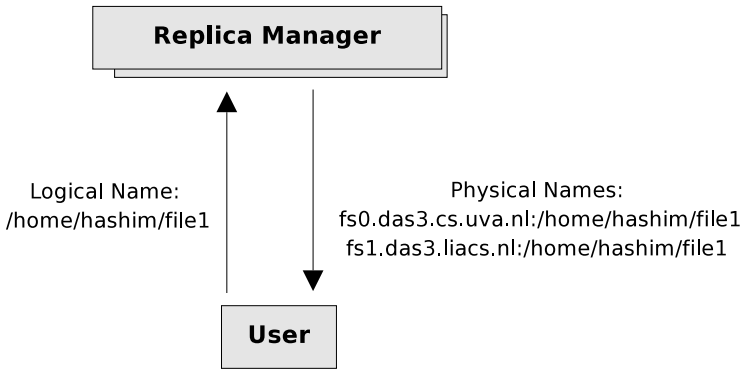



Figure 2.4: Example of the mappings of a logical file name to its physical locations.

### 2.2.3 Job priorities

In real systems, the need may arise to give some jobs preferential treatment over others. For instance, some jobs may have (soft) deadlines associated, or may need interaction from the user. Therefore, we have introduced the priority of a job, which is used to determine its importance relative to other jobs in the system. Currently, we distinguish four priority levels, which are *super-high*, *high*, *low* and *super-low*, and which are assigned to the jobs by our scheduler. Of course, we can have any number of priority levels of jobs, but we have limited the number to only four based on the types of jobs discussed below, which are common in grids:

1. Interactive jobs. These are jobs that run interactively and require quick responses. To avoid delaying these jobs, the super-high priority level is assigned to them.

2. Occasional jobs. These are batch jobs that are submitted specifically for special occasions, such as demos, tight deadlines, etc. We give these jobs a high priority level, and leave the decision of which jobs belong to this group to the system administrator.

3. Batch jobs. These are normal batch jobs that run without any special requirements. We assign these jobs the low priority level.

4. Cycle-scavenging jobs. These jobs scavenge machines for available CPU cycle. Cycle-scavenging jobs can be submitted to both desktop machines and cluster nodes. These jobs are started when CPUs are idle and immediately stopped if the CPUs are needed again by jobs of higher priority levels. In our model, we assign to cycle-scavenging jobs the super-low priority level.

The four priority levels might also be assigned based on a system policy. Examples of these policies include assigning priority levels to jobs based on their estimated job runtimes, with longer jobs having lower priorities. These jobs of different groups of users from different domains or projects can also be assigned priorities based on their importance.

The priority level plays a part during the placement of a job, i.e., when finding suitable pairs of execution sites and file sites for the job components (see Section 4.2.1), and when claiming processors for a job's components (see Section 4.4). During placement, jobs are placed according to their priority levels. Moreover, during claiming processors for jobs, in the absence of processor reservation, it is possible that not enough processors are available anymore. In this scenario, a job of a higher priority is allowed to preempt lower-priority jobs until enough idle processors for it to execute are freed.

## 2.3 Grid Applications

The presence of grid programming models has resulted in an abundance of different grid application types with different and unique characteristics. In Section 2.3.1 we give an overview of different application types that exist, and in Section 2.3.2 we present the sample grid applications that we use in our experiments in this thesis.

### 2.3.1 Grid application types

In computational grids, different application types with different characteristics may exist. Important grid application types include parallel applications, sequential applications, Bags-of-Tasks (BoTs) [72, 102, 105], Parameter Sweep Applications (PSAs) [47, 48, 85], workflows [54, 70, 108], data-intensive applications, and application types written with special grid programming models like Ibis. BoTs, PSAs and workflows are formed by coupling together multiple grid applications based on some rules. A BoT application is composed of independent tasks which can be scheduled and executed in any order without needing inter-task communication. There are many important BoT applications such as datamining, massive searches, Monte Carlo simulations, fractal calculations (such as Mandelbrot), and image processing applications (such as tomographic reconstruction) [105]. PSAs are specialized BoTs with tasks that each execute the same program but

with different parameters. Workflows are concerned with the automation of procedures whereby files and data are passed between participating tasks according to a set of rules to achieve an overall goal [108].

### 2.3.2 Sample grid applications

In this thesis, we have selected the following grid-enabled applications to be used when assessing the mechanisms and policies of our KOALA grid scheduler. Below, we present three MPI applications followed by Ibis applications. More information about these applications including their runtimes will be presented in Section 6.2.1.

#### The Poisson application

This application implements a parallel iterative algorithm to find a discrete approximation to the solution of the two-dimensional Poisson equation (a second-order differential equation governing steady-state heat flow in a two-dimensional domain) on the unit square. For discretization, a uniform grid of points in the unit square with a constant step in both directions is considered. The application uses a red-black Gauss-Seidel scheme, for which the grid is split up into "black" and "red" points, with every red point having only black neighbours and vice versa. In every iteration, each grid point has its value updated as a function of its previous value and the values of its neighbours and all points of one colour are visited first followed by the ones of the other colour. The application is implemented in MPI where the domain of the problem is split up into a two-dimensional pattern of rectangles of equal size among the participating processors. When executing this application on multiple clusters, this pattern is split up into adjacent vertical strips of equal width, with each cluster using an equal number of processors.

#### The Fiber Tracking application

The Fiber Tracking (FT) application uses the anisotropic diffusion of water molecules in the human brain to visualize the white-matter tracts and the connecting pathways between brain structures. Combined with functional MRI, the information about white-matter tracts reveals important information about neuro-cognitive networks and may improve the understanding of the brain function. The FT works by starting at various points, and tracks the white-matter fibers in the entire data domain. The number of detected fibers (and therefore the accuracy of the algorithm) grows with the number of starting points considered. The execution time of this application depends on the number of starting points, the algorithm, and the size of the data set, and can amount to many hours. Therefore, in order to increase the throughput without decreasing the accuracy of the result, the parallel version of this application is used [41], which has been written with MPI. In our

experiments, we use the version of this application compiled with MPICH-g2. The FT application is a proprietary application, which is available to us only in a binary format. As a result, an analysis of its characteristics such as its communication pattern is impossible to us.

### The Lagrangian Particle Model application

The Lagrangian Particle Model (LPM) application has been developed and extensively used by the Department of Applied Mathematical Analysis at Delft University of Technology. This application performs simulation of sediment transport in shallow waters using a particle decomposition approach [14]. The model can also be used to predict through simulations the dispersion of pollutants in shallow waters. In the simulations, the computation cost becomes high because a large number of particles is required. Fortunately, particles behave independently from one another, thus allowing the application to use parallel processing to reduce the runtime.

The simulation of sediment transport is done by numeric integration of a set of stochastic differential equations (SDEs). The displacement of the position of the particles is done by a deterministic part and the random term of the SDEs. This technique of following the track of the sediment particles along their paths in time is known as the Lagrangian particle approach. In our experiments with this application, realistic data of the Dutch coastal waters, notably in the Wadden Sea, were used for the prediction of both sediment transport and pollutant dispersion.

The LPM application is an MPI application, which has been compiled with MPICH-G2 and which can therefore use co-allocation. Compared with the other applications presented in this section, the LPM application is communication intensive with many "many-to-many" communication patterns.

### The Ibis applications

We select three Ibis applications that use either the Satin or the RMI programming model; these programming models have already been described in Section 2.1.8. The Satin applications that we use are N-Queens, which solves the combinatorially hard problem of placing N queens on a chess board such that no queen attacks another, and Raytracer, which computes a picture using an abstract description of a scene. N-Queens uses recursive search to find a solution. On the other hand, Raytracer uses parallel processing to recursively divide the picture up to the pixel level. Next, the correct color of that pixel is calculated and the image is reassembled. The RMI application that we use is red/black Successive Over Relaxation (SOR), which is an iterative method for solving discretized Laplace equations. This application distributes a matrix row-wise among the processors.

Each processor exchanges one row of the matrix with each of its neighbours at the beginning of each iteration.

## 2.4 Related Work

In this section we review the research on resource management and job deployment frameworks in grids that is related to our work. In Section 2.4.1 we consider papers that have studied co-allocation with simulations and co-allocation with advance reservations. Since we deal also with data in our scheduling policies, we also summarize some of the work on data scheduling in Section 2.4.2. Finally, some existing grid resource management systems and grid job deployment frameworks are discussed in Section 2.4.3 and in Section 2.4.4, respectively.

### 2.4.1 Study of co-allocation with simulations

Recent research in computational grids has studied the co-allocation problem in grids with a focus on processor co-allocation only (without considering data). The focus of research has also been on studied different approaches of guaranteeing the simultaneous availability of resources in multiple sites at job start times. In this section we review these works that address the co-allocation problem by means of simulation. These simulation studies are compared to our experiments in Section 5.3, after we have presented the results of our experiments.

Bucur et al. [34, 38, 40] study processor co-allocation in multiclusters with space sharing of rigid jobs for a wide range of such parameters as the number and sizes of the job components, the number of clusters, the service-time distribution, and the number of queues in the system. The main results are that co-allocation is beneficial as long as the number and sizes of job components, and the slowdown of applications due to the wide-area communication, are limited.

The impact of wide-area communication on the efficiency of co-allocation is also studied with simulations by Ernemann et al. [59, 60]. There, jobs only specify a total number of processors, and are split up across the clusters. Co-allocation is compared to keeping jobs local and to only sharing load among the clusters, assuming that all jobs fit in a single cluster. One of the most important findings is that when the application slowdown does not exceed 1.25, it pays to use co-allocation.

An Availability Check Technique (ACT), which is designed to be a complementary technique to most resource co-allocation protocols, is introduced by Azougagh et al. [32]. In this technique, each job gets informed of the state changes of the requested resources until they become available. Once the resources are available, a job applies the selected resource co-allocation protocol to acquire the resources. In this paper, two such protocols,

All-or-Nothing (AONP) and Order-based Deadlock Prevention (ODP$^2$), are used with the ACT. In AONP, a resource co-allocator releases all the resources already allocated if the allocation of at least one of the required resources fails. ODP$^2$, which is proposed to prevent deadlock and to reduce the degree of starvation of resources, requires each job to secure its resources one by one according to a given global order. This means that the distinct resources need to be globally ordered. The results of the simulations show the benefit when ACT is used with the two protocols.

Röblitz et al. [86,87] present an algorithm for reserving compute resources that allows users to define an optimization policy if multiple candidates match the specified requirements. An optimization policy based on a list of selection criteria, such as end time and cost, ordered by decreasing importance. For the reservation, users can specify the earliest start time, the latest end time, the duration, and the number of processors. To allow elasticity in the processor type, a duration is defined for a specific number of reference processors. The algorithm adjusts the requested duration to the actual processor types and numbers by scaling it according to the speedup, which is defined using speedup models or using a database containing reference values. This algorithm supports so-called fuzziness in the duration, the start time, the number of processors, and the site to be chosen, which leads to a larger solution space. This work is presented as a building block for future work that is to provide co-reservations, i.e., the reservations of multiple independent resources.

Smith et al. [90] study a reservation mechanism for restartable and non-restartable applications through simulations. In their work, reservation for an application is made by a scheduler which first simulates the scheduling of applications in the system and produces a timeline of when the processors will be used in the future. This timeline is then used to decide when a reservation can be made. The runtime information required for this mechanism when scheduling applications can be obtained directly from the users, from historical information of the runs of the application, or by running a benchmark application. This mechanism is simple and straightforward, but it depends on the correct predictions of the runtimes of the applications; obtaining these may result in a high scheduling overhead, and they may still be inaccurate.

Azzedin et al. [33] propose the scheme Synchronous Queuing (SQ) for co-allocation that does not require advance reservations. This scheme ensures that the subtasks of a job remain synchronized by minimizing the co-allocation skew, which is the time difference between the fastest running and the slowest running substasks (job components) of an application. Despite a similar aim with KOALA of achieving co-allocation without advance reservations, some differences can be observed. Firstly, this work is only aimed at multimedia applications with long execution times and without sub-task communication. Secondly, synchronization is maintained throughout the execution of the application. In KOALA, only the component (subtask) start times are synchronized, and further synchronization during the execution, if required, is left to the application.

### 2.4.2 Data-aware scheduling

Data intensive applications are common in many disciplines of science and engineering. Such applications can benefit from a grid environment provided that good data-aware scheduling policies that schedule both processors and data are available. Thain et al. [94] propose a system that links jobs and data by binding execution and storage sites into I/O communities that reflect the physical reality. A job requesting particular data may be moved to a community where the data are already staged, or data may be staged to the community in which a job has already been placed. Other research on data access has focused on the mechanisms for automating the transfer of and the access to data in grids, e.g., in Globus [22] and in Kangaroo [93], although there less emphasis is placed on the importance of the timely arrival of data.

Ranganathan et al. [84] discuss the scheduling of sequential jobs that need a single input file in grid environments with simulations of synthetic workloads. Every site has a Local Scheduler, an External Scheduler (ES) that determines where to send locally submitted jobs, and a Data Scheduler (DS) that asynchronously, i.e., independently of the jobs being scheduled, replicates the most popular files stored locally. All combinations of four ES and three DS algorithms are studied, and it turns out that sending jobs to the sites where their input files are already present, and actively replicating popular files, performs best.

Venugopal et al. [100] present a scheduling strategy which has been implemented within the Gridbus broker [6]. For each job, their algorithm first selects the file site that contains the file required for the job, and then selects a compute resource that has the highest available bandwidth to that file site.

The works discussed in this section focus on ensuring that staging of the large input files does not delay the start of the data-intensive applications that require them. KOALA shares this focus but differs by its requirement of simultaneous staging files in multiple locations when scheduling data-intensive applications.

### 2.4.3 Grid scheduling systems

Despite the existence of a number of grid schedulers, to the best of our knowledge, none of them combine processor and data co-allocation in grids without relying on the support of advance processor reservation in LRMSs. Czajkowski et al. [51] propose a layered co-allocation architecture which addresses the challenges of grid environments by providing basic co-allocation mechanisms for the dynamic management of separately administered and controlled resources. These mechanisms are implemented in a co-allocator called the Dynamically Updated Resource Online Co-allocator (DUROC), which is part of the Globus project [22]. DUROC implements co-allocation specifically for the grid-enabled implementation of MPI applications. However, DUROC, which is implemented as a set

of libraries to be linked with application codes and job submission tools, does not provide resource brokering or fault tolerance, and requires jobs to specify exactly where their components should run. As a consequence, DUROC only supports fixed job requests. In our work, we use DUROC as a building block of the component of KOALA that enables the co-allocation of MPI applications.

The Globus Architecture for Reservation and Allocation (GARA) [64] enables the construction of application-level co-reservation and co-allocation libraries that are used to dynamically assemble a collection of resources for an application. In GARA, the problem of co-allocation is simplified by the use of advance reservations. Support for co-allocation through the use of advance reservations is also included in the grid resource broker presented by Elmroth et al. [58]. The limited support of advance reservations by LRMSs hinders the wide deployment of co-allocation mechanisms that do depend on such reservations. With KOALA, we address this limitation and we have implemented a work-around mechanism for advance processor reservations.

Nimrod-G [27] is an economy-driven grid resource broker that supports soft-deadline and budget-based scheduling of applications on the grid. Like KOALA, Nimrod-G performs resource discovery, scheduling, dispatching jobs to remote grid nodes, starting and managing job execution, and gathering results back to the submission site. However, Nimrod-G uses user-defined deadline and budget constraints to make and optimize its scheduling decisions, and focuses only on parameter sweep applications.

The GridWay framework [68], which allows the execution of jobs in dynamic grid environments, incorporates similar job scheduling steps as KOALA does, such as resource discovery and selection, job submission, job monitoring and termination, but then at the application level (Application Level Scheduling). An important drawback of GridWay is that the number of applications that can be run with GridWay is limited because application source code first needs to be instrumented to employ the framework. Another drawback of this system is that its scheduling process is not aware of other jobs currently being scheduled, rescheduled, or submitted, which has as a consequence, a degradation of the throughput of the grid. With KOALA, scheduling is performed at the system level by a scheduler, that has knowledge of the entire system, which helps to maximize the throughput of the system. The KOALA job submitters (runners) can perform application level scheduling of the resources allocated to them by the KOALA scheduler.

The Grid Application Development Software (GrADS) [52] enables co-allocation of grid resources for parallel applications that may have significant inter-process communication. For a given application, during resource selection, GrADS first tries to reduce the number of workstations to be considered according to their availabilities, computational and memory capacities, network bandwidth, and latency information. Then, the scheduling solution that gives the minimum estimated execution time is chosen for the application. Like GridWay, GrADS performs only application level scheduling and there-

fore, shares the same limitations with respect to KOALA.

Condor with its DAG-manager is a system that is able to perform allocation of resources in different administrative domains to a single job [66, 95]. Condor's DAGMan takes as input job descriptions in the form of Directed Acyclic Graphs (DAGs), and schedules a task in such a graph when it is enabled (i.e., when all its precedence constraints have been resolved). However, no simultaneous resource possession as part of a co-allocation mechanism is implemented. Raman et al. [83] have extended the Condor class-ad matchmaking mechanism for matching single jobs with single machines to "gangmatching" for matching single jobs with sets of resources, which amounts to co-allocation. The running example in their work is the inclusion of a software license in a match of a job and a machine, and it was promised that the gangmatching mechanism will be extended to the co-allocation of processors and data.

The Community Scheduler Framework (CSF) is an open-source implementation of a number of grid services, which together perform the functions of a grid metascheduler [49, 106]. CSF is built on top of the Globus Toolkit 4.0 and therefore, it is limited to grids that use the Globus middleware. CSF consists of a number of web services such as, a Job Service, a Reservation Service, a Queuing Service, and Resource Manager Services. The Job Service provides the interface for end users to fully control their jobs, while the Reservation Service allows the end users to reserve resources for their jobs in advance. Since CSF relies on Globus for middleware services and at the time of writing of this thesis GRAM did not support resource reservation, the reservation requests are simply forwarded to the LRMS. Again, this reservation mechanism is limited to only clusters with LRMSs that support advance processor reservations. The Queuing Service of CSF represents a set of scheduling policies and associated job requests. Queues start to schedule jobs periodically only in a FCFS manner. It should be noted that KOALA is not tied to one grid middleware, and that our modular design allows KOALA to operate on top of any grid middleware as will be discussed in Chapter 3.

### 2.4.4 Grid job deployment frameworks

In computational grids, different application types with different characteristics forming complex workloads exist. The automation of submission mechanisms for these workloads to the grid infrastructure is far from complete, and job deployment still requires specialized operation skills. Attempts have been made to solve this problem for different application types. Euryale [101] is a system designed to run jobs over large grids. Euryale uses Condor-G [66] to submit and monitor jobs, and it takes a late binding approach in assigning jobs to sites. It also implements a simple fault tolerance mechanism by means of job re-planning when a failure is discovered. Euryale can be integrated in different grid schedulers like our KOALA, and be used as a job submission tool for running jobs

with Condor-G. Condor-G's DAGMan executes Euryale's prescripts and postscripts. The prescript of a job calls the external site collector such as the KOALA scheduler to identify the site on which the job should run. The postscript transfers output files to the collection area and inform the monitoring tools.

The grid programming environment called ASSIST [53] aims to offer grid programmers a component-oriented programming model, in order to enforce the reuse of already developed code. In the same work a grid execution agent that performs resource discovery, resource selection and mapping, file staging, and launching the application for execution is presented. Clearly, this execution agent combines the functionality of the scheduler and the submission tool into one monolithic structure which makes its extension to more application types difficult.

In the AppLeS project [48], each grid application is scheduled according to its own performance model, which is provided by the user. The general strategy of AppLeS is to take into account resource performance estimates to generate a plan for assigning file transfers to network links and tasks (sequential jobs) to hosts. This functionality can be achieved in the KOALA framework proposed in this thesis by means of a runner. An example of this is a new runner called the MDrunner, which was written specifically for Higher-Order Component (HOC) applications [55]. This runner first requests a number of execution sites from the KOALA scheduler and then organizes the execution of a job's components on these sites based on resource performance.

# Chapter 3

# The KOALA Grid Resource Management System

In this chapter we describe the design of the KOALA grid resource management system and our experiences with it. With KOALA, we try to meet the challenges of resource management and jobs deployment in grids presented in Chapter 1. KOALA is designed for multicluster systems like the DAS that have in each cluster a head node and a number of compute nodes. KOALA started as a prototype named the Processor and Data Co-Allocator (PDCA) [77–79], and has been in operation in the DAS-2 system since September 2005. In May 2007, KOALA was also ported to the DAS-3 [11].

KOALA has a layered architecture that allows us to develop distinct layers independently, which can then work together. The KOALA layered architecture consists of four layers: the *scheduler*, the *runners framework*, the *runners*, and the *submission engines*, as shown in Figure 3.1. The KOALA scheduler, which is the subject of Section 3.1, is responsible for scheduling jobs received from the runners. The scheduler is equipped with placement policies that are used to place jobs on suitable execution sites, and a claiming policy that is used to claim for jobs their assigned processors at their scheduled times. The choice of which placement policy to use is initiated by the runners and therefore, it can be selected by the users for every submitted job separately. The runners framework presented in Section 3.2 hides the heterogeneity of the grid by providing to the runners a runtime system and its corresponding set of APIs for commonly used job submission operations. The runners are specialized KOALA components for submitting and monitoring different applications types; they will be discussed in Section 3.3. The implementation details of the runners framework and the runners are presented in Section 3.4. The last layer consists of the submission engines, which are third-party tools that use the runners to submit jobs to KOALA. These tools include workflow engines and workload generation and submission tools. This layer is explained in Section 3.5. A layered approach benefits from the advantages of modularity and flexibility. Also, it is common practice to use

a layered architecture to separate the scheduler and the job submitters; however, our job submitters (runners) need to satisfy the challenge of deploying grid applications presented in Section 1.2, hence the introduction of the runners framework.

Jobs are guided through the layers of KOALA according to the KOALA job flow protocol, which is the subject of Section 3.6. In our experiences with deploying KOALA on the DAS, users care much more about their jobs correctly finishing than about the performance of the their jobs. In Section 3.7, we discuss the reliability of KOALA. The DAS testbed uses the Globus Toolkit as its grid middleware; our experiences of using Globus on the DAS are the subject of Section 3.8. Finally, in Section 3.9 we discuss the wide range of usage of KOALA in the past two years by different users of the DAS.

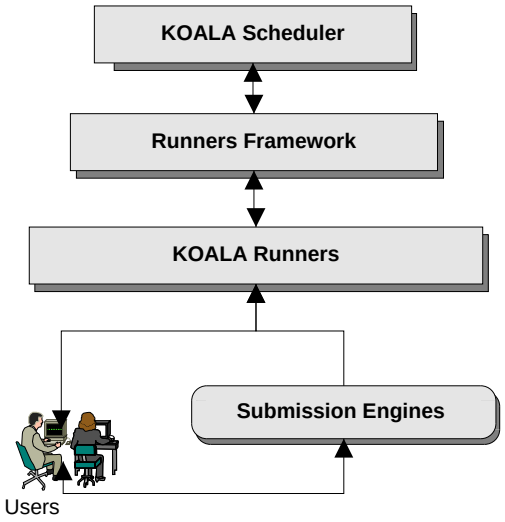



Figure 3.1: The KOALA layered architecture.

## 3.1 The KOALA Scheduler

The KOALA scheduler is responsible for scheduling jobs received from the KOALA runners or any third-party job submission tools. By scheduling we mean deciding where and when the components of a job should be sent for execution. The scheduler uses one of its placement policies to select file sites and execution sites for the job components. By placement we mean finding execution sites for job components with sufficient number of processors. A new job arriving to the scheduler is appended to the tail of one of the placement queues depending on its priority, and its placement is retried for a fixed number of times. The scheduler also decides when the job components should start executing. If

processor reservation is supported by the local resource managers, the KOALA scheduler reserves processors for the components, otherwise the KOALA claiming policy is used for claiming processors for the job components at their designated times possibly in multiple sites. If the claiming procedure of processors for the job components fails, the job is added to the claiming queue and the claiming is retried. In Section 3.1.1 we describe the scheduler components, and in Section 3.1.2 we discuss some details of the implementation of the scheduler. The placement queues and the claiming queue are presented in Sections 4.2.1 and 4.2.2, respectively, and the placement policies and the claiming policy in Sections 4.3 and 4.4, respectively.

### 3.1.1 The scheduler components

The KOALA scheduler consists of the following three components: the *Co-allocator* (CO), the *Information Service* (IS), and the *Processor Claimer* (PC). The structure of the KOALA scheduler is depicted in Figure 3.2. We will now discuss these components in turn.

The CO is responsible for placing jobs, i.e., for finding the execution sites with enough idle processors for their components. The CO chooses jobs to place based on their priorities from one of the KOALA placement queues. If the components require input files, the CO also selects the file sites for the components such that the estimated file transfer times to the execution sites are minimal. To decide on the execution sites and file sites for the job components, the CO uses one of the placement policies discussed in Section 4.3. Finding execution sites for the job components is only done for non-fixed, semi-fixed, and flexible job requests.

The IS is comprised of the Replica Information Provider (RIP), the Network Information Provider (NIP), and the Processor Information Provider (PIP). The RIP provides to KOALA a common interface for mapping the logical file names to their physical locations. The RIP can use the Globus Toolkit's Replica Location Service (RLS) [22] to provide this mapping if present. In the absence of the RLS, RIP maintains a simple database that can be used to provide the mapping. The NIP provides to KOALA a common interface to obtain network bandwidth and latency measurements from third-party tools. At the moment, in the DAS *Iperf* [10] is used for this purpose. A repository containing the bandwidths measured with NIP is maintained and updated periodically.

The PIP, like the NIP, provides to KOALA a common interface to access grid information providers like the Globus Toolkit's Metacomputing Directory Service (MDS) [22] to obtain status information of nodes per cluster like the numbers of busy nodes and erroneous nodes, and the total number of available nodes. The PIP also has native mechanisms to query the status information of nodes straight from local resource managers (currently,

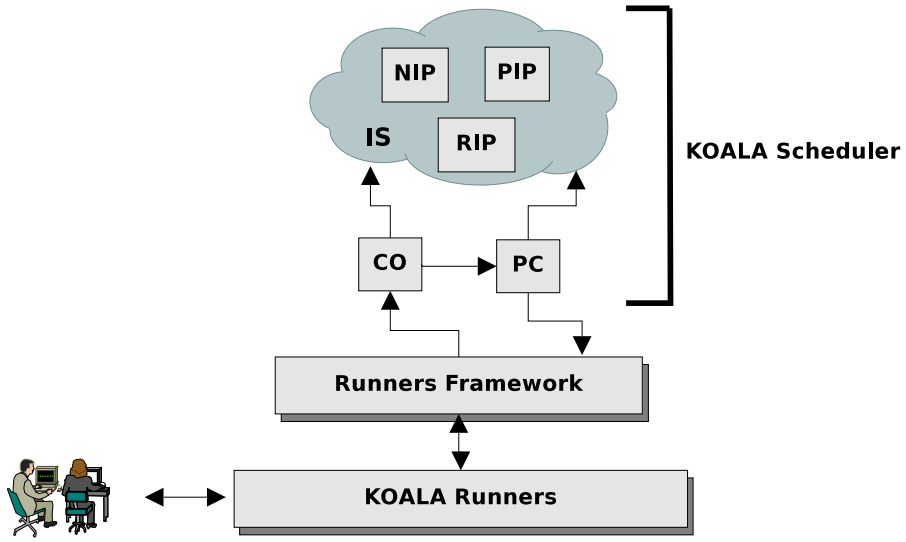



Figure 3.2: The components of the KOALA scheduler.

only PBS and SGE are supported). The native mechanisms are useful where the MDS is missing, or when MDS shortcomings are observed. The shortcomings are felt in busy systems where Globus MDS-2 suffers from a number of problems, including the fact that updated information does not propagate very quickly and that centralized servers may become bottlenecks or points of failure [82]. It should be noted that requests for node status and requests to the bandwidth repository impose delays on placing jobs, especially on busy systems. Therefore, to minimize the delay, the IS caches information obtained from the PIP and the NIP with a fixed cache expiry time (a parameter of KOALA). Furthermore, to deal with the fact that updated information may not always propagate quickly when requested, the IS can be configured to do periodic cache updates from frequently used clusters before their cache expiry time.

After a job has been placed, it is the task of the PC to ensure that processors will still be available when the job starts to run. If processor reservation is supported by local resource managers, the PC can reserve processors immediately after the job placement. Otherwise, the PC uses KOALA's claiming policy to postpone claiming of processors to a time close to the estimated job start time; this policy is discussed in detail in Section 4.4.

### 3.1.2 The implementation of the scheduler

The KOALA scheduler is implemented using Java 5.0. The Java language was chosen because it solves many software portability issues present in the heterogeneous grid environment. Also, the choice to use Java was a result of the lessons learnt with the PDCA, the precursor of KOALA, which was written entirely in C. The development and maintenance

cost of the PDCA was very high compared to the value gained by using the language C, which was the speed with which the scheduler was running. However, most current Java VM implementations come with good state-of-the-art Just In Time (JIT) technology that makes its interpreted code run at compiled code speeds, and therefore, fast enough.

We have ensured that the scheduler is not tied to any particular operating system or grid middleware by using the plugin technology for the IS component. In this component, plugins for the RIP, the PIP, and the NIP specific for a grid middleware or an operating system can be added and then registered to be used by the KOALA scheduler. The same technology is used for the placement and claiming policies, where new policies can be added and then registered to the CO without affecting its operation. Examples of policies that have been added in this manner are the Cluster Minimization (CM) and the Flexible Cluster Minimization (FCM) policies [91], which are more application oriented. The choice of which policy to use is initiated by the runners and therefore, it can be selected by the users for every submitted job separately.

## 3.2 The KOALA Runners Framework

Different types of grid applications have different characteristics that pose varying difficulties when attempting to deploy them on grids. The challenges posed by grid applications and the rate at which grid technology is changing makes it impossible to have one universal submission tool, a *runner*, for all current and future grid applications. Therefore, we introduce our runners framework, which not only hides the heterogeneity of the grid, but also allows easy addition of new runners for new application technologies or modifying existing ones with minimal effort [76]. Frameworks are an effective tool to deal with the complexity of grid applications and heterogeneity of current computing environments, and are an important insurance policy against disruptive changes in future technologies. Our runners framework provides a set of APIs to the runners for commonly used grid job submission operations such as interfacing with the KOALA scheduler for job scheduling, the transfer of input files, deploying jobs on grids, monitoring and responding to failures, and the transfer of output files back to the submission site. Also, the runners framework enforces the KOALA job flow protocol that needs to be followed by all jobs submitted through KOALA, and that is described in Section 3.6. In Section 3.2.1 we present the components of the runners framework and in Section 3.2.2 we describe the fault tolerance mechanisms of this framework.

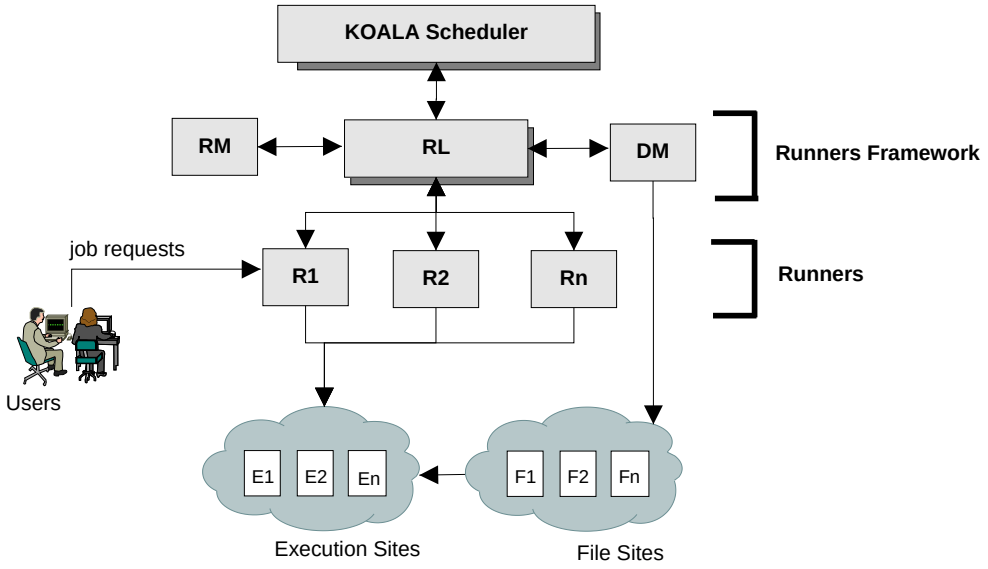



Figure 3.3: The runners framework of KOALA.

### 3.2.1 The framework components

The runners framework provides a set of APIs to the runners for commonly used grid job submission operations such as interfacing with the KOALA scheduler for job scheduling and interfacing with a grid middleware for deploying jobs on grids, monitoring job submission failures, and transferring input and output files. Based on these operations, the runners framework consists of the following three components: the *Runners Listener* (RL), the *Run Monitor* (RM), and the *Data Manager* (DM). The framework is depicted in Figure 3.3.

The RL acts as the interface between the runners and the runners framework. It provides to the runners a set of APIs for job scheduling with the KOALA scheduler, and for application level operations such as input and output file transfers, deploying jobs on grids, and monitoring and responding to failures. The runners have complete freedom to implement their own mechanisms for the application level operations, or alternatively, to use the default implementations that are provided by the runners framework. The implementation of the application level operations depends on the grid middleware in use, and currently, only the Globus middleware is supported.

The KOALA job flow protocol guides a job from the moment it is received from a user until its completion time. When a new job request is received from a user by a runner, of which a separate instantiation is created for every job, the RL on behalf of the runner asks the scheduler to schedule the job. If the job requires input files and after instructed to do so by the scheduler, the RL invokes the file transfer mechanisms supplied by the runner or the DM by default to transfer the input files to the execution sites. At the time of claiming

the processors for the job components, the scheduler sends the components back to the RL for submission to their respective execution sites. During the execution of the job, the RL may receive instructions from a runner to abort and/or re-schedule job components. The KOALA job flow protocol will be described in more detail in Section 3.6.

The RM implements the mechanisms to launch the job components for execution, and to monitor and respond to errors that may interrupt the execution of any of the job components. The implementation of the RM depends on the grid middleware in use. The current implementation of the RM uses the Globus Resource Allocation Manager (GRAM) to launch the job components to their respective execution sites. The mechanisms for responding to errors are discussed in Section 3.2.2.

The DM is used to manage file transfers for which the current implementation uses both Globus GridFTP [28] and Globus Global Access to Secondary Storage (GASS) [22]. The DM is responsible for ensuring that input files arrive at their destinations before the job starts to run.

The runners provide to the user an interface for submitting their job requests and for monitoring the progress of the job execution. The KOALA runners are discussed in Section 3.3.

### 3.2.2 Fault tolerance

The RM monitors the execution of the components of a job for errors that may interrupt their execution. These errors, which are hardware, operating-system, grid-middleware, or application related, are divided into three groups depending on whether they are system errors occurring at (the head node of) the submission site of the job, whether they are system errors at one of the execution sites of a job, or whether they are related to the application itself.

The first group of errors we distinguish in the context of the execution of a job are irrecoverable *hard errors*, which are caused by operating-system and network failures of the submission site of the job, i.e., at the head node of the site where the runner of the job has been launched, of which are grid middleware errors, e.g., middleware software bugs. The RM responds to errors of this group by informing the responsible runner that it is going to abort the job, and also as a consequence, the runner itself, before performing this action.

The second group of errors consists of *soft errors*, which are execution-site specific errors caused by hardware or software faults of the execution sites of a job. These are errors of nodes executing a job component at an execution site, which are reported by the grid middleware. The RM responds to soft errors in the context of the execution of a job by allowing the runner of the job to deal with the job component(s) that have reported such errors, and at the same time, by informing the scheduler about the erroneous clusters.

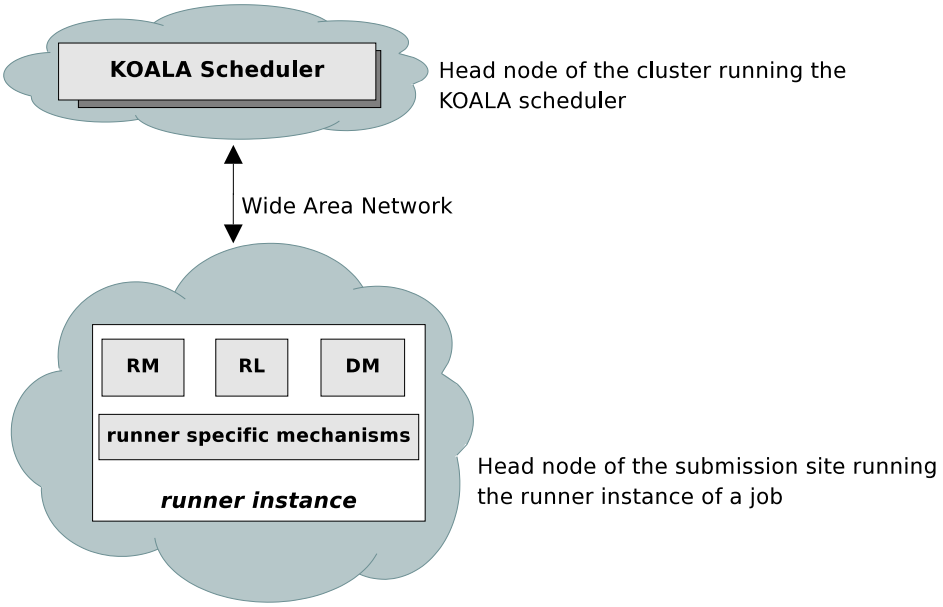



Figure 3.4: An instance of a KOALA runner.

The KOALA scheduler counts the number of consecutive errors of each cluster and if this number reaches a certain threshold, the cluster is marked unusable. A message containing the details of the error is then sent to the administrator of the system for further action. If a runner does not have mechanisms to deal with soft errors, the default operation in the RM is to abort the job, to inform the scheduler to use other available execution sites for the aborted job components, and then to restart the job. It should be noted that the RM focuses only on ensuring that the job is restarted. It is left for the runner to employ application-specific fault tolerance mechanisms like checkpointing to ensure that a job continues its execution from where it left off and does not restart from the beginning.

The third group of errors we distinguish in the context of the execution of a job are *application-specific errors*, which can be anything caused by faults in the application, and which are simply passed back through the RL to the runner of the job. The default operation for these errors is to simply abort both the job and the runner, if no mechanisms to deal with these errors are present in the runner.

## 3.3 The KOALA Runners

The KOALA runners implement specific mechanisms for launching jobs of their respective application types on grids and for monitoring job executions and responding to failures

through the use of APIs in the RL and the RM. Each job submitted to KOALA has its own instance of the runner corresponding to its application type. A runner consists of runner-specific mechanisms, and of instances of the RM, the RL, and the DM. A runner can be used to submit jobs from any site in a multicluster system or a desktop machine and is able to interface to the centralized KOALA scheduler over the wide-area network. Figure 3.4 shows a configuration with an instance of a runner.

In Section 3.3.1 we first present the requirements the runners need to satisfy. Sections 3.3.2–3.3.4 present the runners that are fully operational on the DAS system. All these runners support co-allocation.

### 3.3.1 Requirements to the runners

Job requests of different application types when submitted to KOALA may pose different requirements that need to be satisfied by the runners. Below, we list the major requirements that should be addressed by the runners. It should be noted that these requirements are not exhaustive and can be extended when the need to do so arises.

1. **The placement procedure**. The success of the placement of a job in the KOALA scheduler is specific to the application type. For example, with parallel applications, job placement succeeds only if all of the job components can be placed, i.e., these applications require atomic placement. For application types like PSAs and BoTs with components executing independently of each other, atomic placement is not necessary. With these application types, placement is successful if at least a predetermined number of components (e.g., at least one) have been placed.

2. **Deploying order of the components**. Different application types require different orders of deploying the job components to their respective execution sites. For instance, parallel applications require deploying of the job components to be synchronized. The need for a specific order of deploying the job components is evident in jobs with inter-component dependencies like workflows. With PSAs the order is not relevant.

3. **Application level scheduling**. The runners provide the environment for users to develop per-application schedulers that are specially tailored to match the needs of applications such as PSAs, BoTs and workflows. The runners of these application types ask the KOALA scheduler for execution sites and then map the job components to these execution sites. The mapping of the components to a set of execution sites is guided by application level scheduling policies, which are normally aimed at minimizing the response times of the applications.

4. **Wide-area communication between components**. Wide-area communication is vital to the successful execution of some application types like parallel applications, Ibis applications, and workflows. For parallel and Ibis applications, wide-area communication allows processors on different sites allocated to the same job to exchange messages. Runners of these application types need to know how to link up with the communication libraries for wide-area communication provided by the applications' programming models; for some application types like BoTs, inter-component communication is not required.

5. **Fault tolerance**. Runners need to deal with soft errors and application-specific errors since these are unique to different application types. For some application types like PSAs, failures of components can be tolerated to some degree. Failed components can even be restarted on different sites without affecting the execution of the job. On the other hand, for some application types like parallel applications and workflows, the failure of a single component can cause other components or even the whole application to fail.

### 3.3.2 The KRunner

The KRunner (KOALA default runner) is KOALA's bare-bone runner capable of running application types that do not have any special requirements. Jobs submitted with the KRunner to the scheduler are placed atomically, i.e., the placement procedure only succeeds if all the job components can be placed. When running a multi-component job with the KRunner, each component is executed independently. It is left for the application itself to handle any inter-component communication needed, as well as the synchronization of the execution of its components. The KRunner does support input file and executable staging to the execution site(s) before the job executes, and retrieval of output files when the job completes. The KRunner is also used as the basis for other runners. It gives the basic implementation a runner needs to be able to interface with the KOALA scheduler.

### 3.3.3 The DRunner

The DRunner (DUROC Runner), which is based on the KRunner, is the specialized runner for Message Passing Interface (MPI) jobs. While the DUROC (Dynamically-Update Request Online Co-Allocator) library [22] from the Globus Toolkit implements the allocation operation across multiple sites (co-allocation), it does not provide any form of resource-brokering or fault tolerance. The DRunner together with the KOALA scheduler adds these functionalities to DUROC. The DRunner uses the DUROC library to deploy and synchronize the execution of a co-allocated job's components across the sites specified by the scheduler. At the same time, the DRunner adds to the Globus DUROC

framework the KOALA functionalities of non-fixed, semi-fixed, and flexible job requests (DUROC only supports fixed requests), the ability to schedule job requests using the KOALA placement and claiming policies, and fault tolerance. The MPI jobs to be submitted with the DRunner need to be compiled with the grid-enabled implementation of MPI called MPICH-G2 [17]. MPICH-G2 allows us to couple multiple sites to run a single MPI application by automatically handling both inter-site and intra-site messaging. Like the KRunner, jobs submitted with the DRunner are placed atomically.

### 3.3.4 The IRunner

The IRunner is designed to run Ibis applications, which are applications that use the specialized Ibis Java communication library [98]. To run an Ibis job, a central component called the Ibis nameserver is required to coordinate the setup of communication ports between job components. A single instance of the Ibis nameserver can be shared among several Ibis jobs. The IRunner can start up several nameservers, one per cluster, and balance their usage among jobs of different users. Alternatively, the IRunner can also start a nameserver for every job.

## 3.4 The Implementation of the Runners Framework and the Runners

Like the KOALA scheduler, the runners and the runners framework are written in Java 5.0. The RL is a Java class that is extended by a runner, and so the runner has the freedom to have its own implementation of the APIs of the RL. These APIs include methods for submitting a job component, for stopping a running job component, for transferring files, etc. The default implementations of these APIs are found in the RM and the DM; the runner is free to choose an implementation that is suitable for the grid middleware it wants to run with.

Figure 3.3.4 shows the implementation of a runner that is used as a basis for other runners. The runner starts by parsing the arguments describing a job passed to it by a user and registers the instances of the RM and the DM, which are specific to the grid middleware; in our current implementation, the RM and the DM are specific for the Globus middleware. The runner then starts the main thread in the RL, which is responsible for guiding the operation of the runner and for enforcing the KOALA job flow protocol. This thread first calls the prePhase method in the runner before passing the job request on to the scheduler for scheduling. This method prepares the runner for launching the job by performing such operatins as creating a sandbox for the run, executing a script to bypass a firewall, and, for a runner like the IRunner, for starting a nameserver if required. The

```
public class Runner extends RL {

 public boolean prePhase()
 // this is the optional method that is called with the RL
  // before the job is submitted for scheduling.
 the job is submitted to KOALA
}

public void preComponentSubmission() {
 // this is the optional method that is called with the RL before
  // the component is submitted for execution by the RL.
}

public void postPhase(boolean jobRunWasSuccessful) {
 // this is the optional method that is called with the
  //   RL after the job has finished its execution successfully.
}

public static void main(String[] args) {
 parseParameters(args);
 registerRM(rm);
 registerDM(dm);
 startRL();

}
```

Figure 3.5: The structure of the source code of the runners

main thread then listens for commands from the scheduler, such as commands to transfer files and to submit job components for execution. The registered instances of the RM and DM are called when such commands are received. Before actually submitting a job component for execution, the preComponentSubmission method is called. This method, which is runner specific, gives the runner the ability to prepare the component just before its submission. For example, with the IRunner, the preComponentSubmission is used for adding Ibis runtime class libraries. This method can also be used to synchronize the start of a job's components. For runners that implement application-level scheduling, the pre-ComponentSubmission method is used to collect the processors from all execution sites as allocated by the scheduler and to reschedule them. Finally, the postPhase method is called when a job has just finished; one of the tasks that can be implemented in this method is

cleaning up.

## 3.5 The Submission Engines

Submission engines are third-party tools that can be used to submit jobs to KOALA. Submission engines such as workflow engines, workload generation and submission engines, and user scripts, use the runners to do the actual job submission to the grid resources. We have tested KOALA with Karajan [70], which is an abstract workflow engine describing a workflow in an abstract form without referring to specific grid resources for component execution (non-fixed requests). In this way, an abstract model provides a flexible way for users to define workflows without being concerned about low-level implementation details. In an abstract model, components can be mapped onto any grid sites with our KOALA scheduler. We have not tested KOALA with other types of workflow engines such as concrete and dynamic workflow engines.

Grenchmark [5,69] is a framework for synthetic workload generation that uses KOALA for job submission. The workload generator of Grenchmark is based on the concepts of so-called unit generators and of Job Description File (JDF) printers. The unit generators produce detailed descriptions for running a set of applications (workload unit), according to the workload description provided by the user [69]. In principle, there is one unit generator for each supported application type. This makes Grenchmark extensible as new unit generators for new application types can be added with ease. The JDF printers take the generated workload units and create KOALA job description files for every job in the units, which are to be submitted to KOALA with the runners.

## 3.6 The KOALA Job Flow Protocol

The KOALA job flow protocol guides jobs through the four phases that any job submitted to KOALA goes through. These phases, which are shown in Figure 3.6, are formed by the four operations that are performed to a job during its life cycle in KOALA. In Section 3.6.1 we give an overview of the four phases and in Section 3.6.2 we describe them in detail in terms of the interaction of the components of KOALA in these phases.

### 3.6.1 The KOALA operational phases

Four operations are performed to any job submitted to KOALA, which are placing its components, transferring its input files, claiming processors for its components, and launching it and monitoring its execution. These operations, which form four phases that the job undergoes, are shown in Figure 3.6. In phase 1, a new job submitted to KOALA is appended

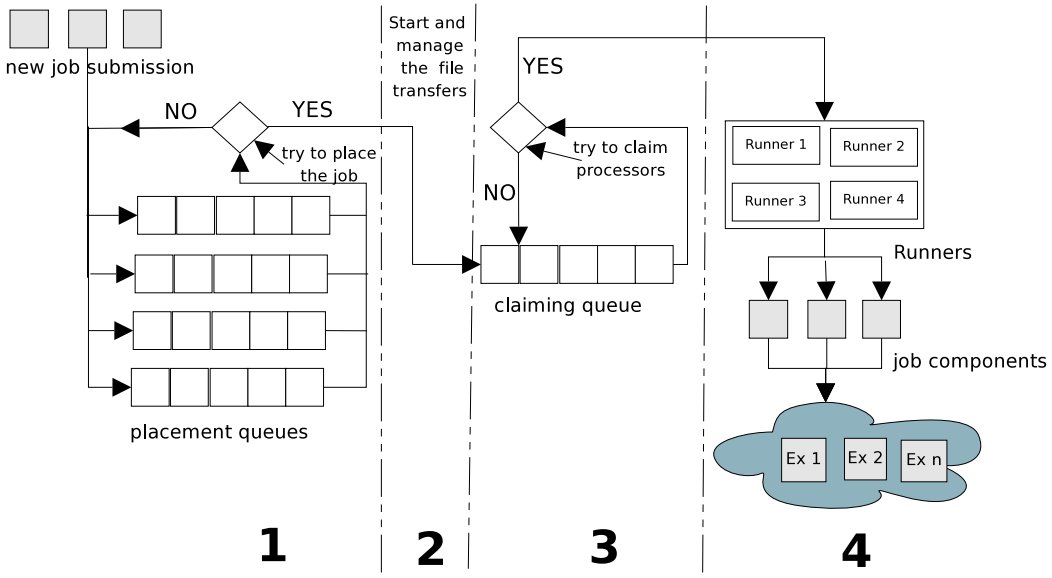



Figure 3.6: The four phases of the job flow protocol in KOALA.

to the tail of one of the placement queues depending on its priority. When it is its turn, the job is retrieved from its placement queue and the placement of its components on the system is attempted. If the placement procedure fails, the job is simply returned to its respective placement queue. The placement of the job is tried again at later times for a fixed number of times. Phase 2 is composed of starting and managing the file transfers for the job if it has been placed successfully in phase 1. It is in this phase that the estimation of the start time of the job is made to ensure that the input file transfers are completed before the job execution starts. As soon as the file transfers are initiated, the job is added to the claiming queue. Jobs are moved immediately to phase 3 in case no file transfers are required. In phase 3, while the job is in the claiming queue, attempts to claim processors for the job components are made at designated times. The job is in phase 4 if all of its components have been launched on their respective execution sites after the success of claiming in phase 3.

### 3.6.2 KOALA components interaction

A number of interactions between the KOALA components occur in each of the phases presented in Section 3.6.1. Figure 3.7 shows these interactions as a job moves from one phase to another. The arrows in this figure correspond to the description given below of the interactions happening in each phase.

In phase 1, a new job request arrives at one of the runners (arrow 1 in Figure 3.7) in the form of a Job Description File (JDF). We use the Globus Resource Specification

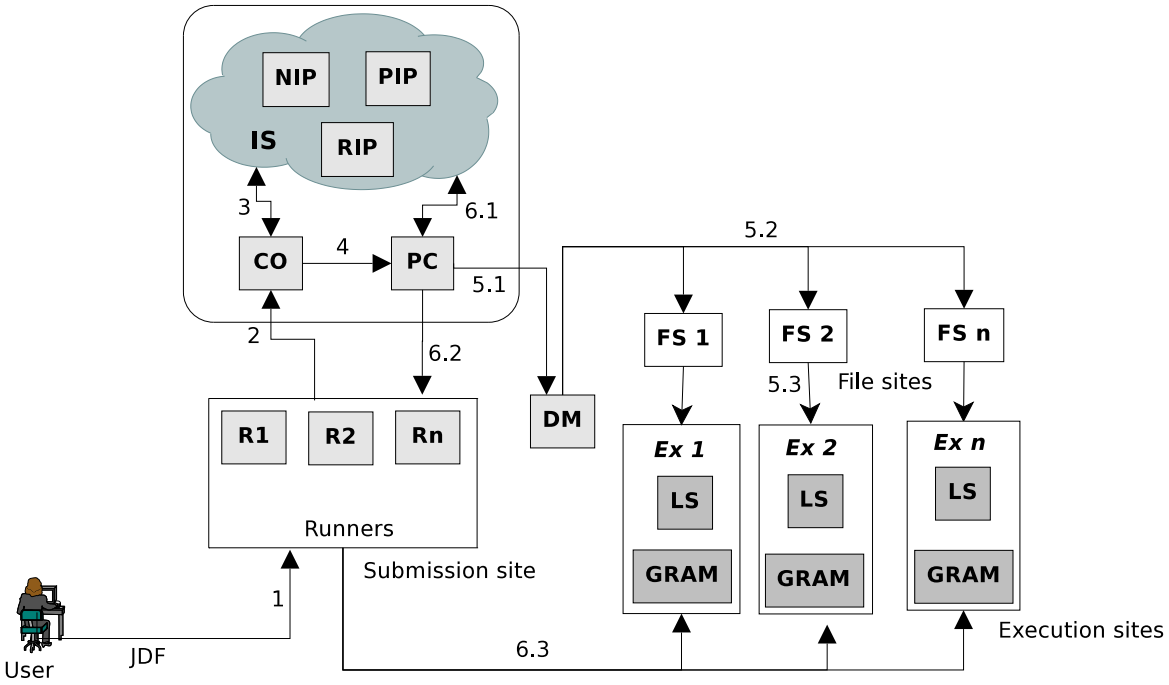



Figure 3.7: The interaction between the KOALA components. The arrows correspond to the description in Section 3.6.2.

Language (RSL) [22] for JDFs, with the RSL "+" construct to aggregate the components' requests into a single multi-request. After authenticating the user, the runner submits the JDF to the CO (arrow 2), which in turn will append the job to the tail of one of the KOALA placement queues. The CO then retrieves the job from this queue and tries to place the job components based on information (number of idle processors and bandwidth) obtained from the IS (arrow 3). If the job placement fails, the job is returned to its respective placement queue. The placement procedure will be tried for the jobs in the placement queues at fixed intervals for a fixed number of times. The placement queues are discussed further in Section 4.2.1.

Phase 2 starts by the CO forwarding the successfully placed job to the PC (arrow 4). On receipt of the job, the PC estimates the Job Start Time and then instructs the DM (arrow 5.1) to initiate the third-party file transfers (arrows 5.2) from the file sites to the execution sites of the job components (arrows 5.3). A detailed description of the timeline of a job defining its Job Start Time and its Job Claiming Time can be found in Section 4.1

In phase 3 the PC estimates the appropriate time that the processors allocated to a job can be claimed, which is called its Job Claiming Time (see Section 4.1). At this time and if processor reservation is not supported by the local resource managers, the PC uses a claiming policy to determine the components that can be started based on the information from the IS (arrow 6.1). It is possible at the Job Claiming Time for processors not to be

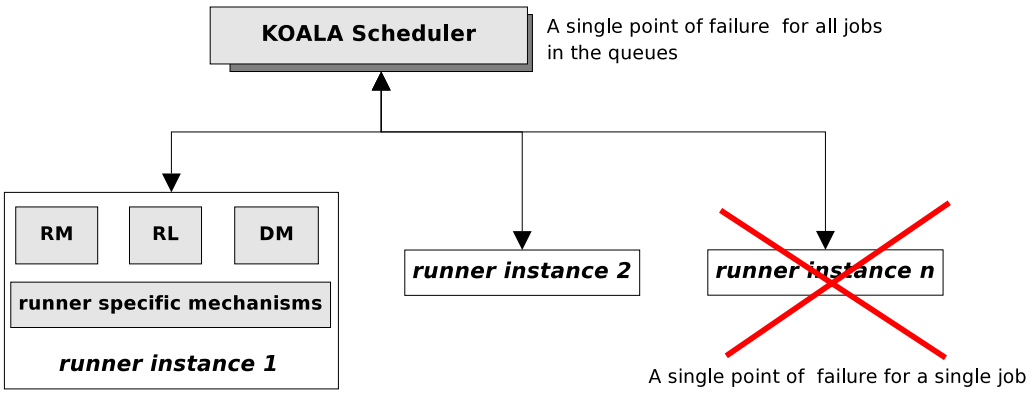



Figure 3.8: The reliability of KOALA.

available anymore, e.g., they can then be in use by local jobs. If this occurs, the claiming procedure fails, the job is put into the claiming queue, and the claiming is tried again at a later time.

In phase 4, the runner used to submit the job for scheduling in phase 1 receives the list of components that can be started (arrow 6.2) and forwards those components to their respective execution sites (arrows 6.3). At the execution sites, the job components are received by a grid middleware component such as the Globus Resource Allocation Manager (GRAM), which is responsible for locally authenticating the owner of the job and sending the job component to the local resource manager for execution.

## 3.7 The Reliability of KOALA

In grid research, performance usually takes a prominent place, but in our experiences with deploying KOALA on the DAS, it is reliability that is the first consideration—users care much more about their jobs correctly completing than about squeezing the next 10% reduction in response times from the system. Prior to the deployment of KOALA on the DAS, we have spent a large effort in making KOALA reliable. Much of this effort went into testing and debugging, but there are also some design considerations that went into making KOALA reliable enough to be released for general use on the DAS. The reliability of KOALA can be discussed from two angles: the reliability of the scheduler, and the reliability of the runners. Figure 3.8 illustrates the reliability issues of KOALA when running on a multicluster system.

When designing KOALA, we opted for a centralized scheduler despite the potential reliability issue of a centralized solution being a single point of failure. The reason for this choice was that the size of the DAS with only five clusters was too small to really warrant a decentralized solution. This is about to change with the ongoing effort to connect the

DAS-3 with Grid'5000, which has already resulted in discussions about having multiple KOALA schedulers running in a decentralized fashion [43]. It should be noted however, that even in the DAS, nothing prohibits the installation and deployment of multiple instances of KOALA. The only thing is that they may reduce each other's performance when they try to allocate and claim the same processors. On the other hand, the centralized approach on the DAS has proven to be very reliable, and in the last two years, that is, since the initial deployment of KOALA, the only restarts of the scheduler were due to the reboots of the DAS cluster where it runs. Even if the scheduler would fail, this would only have consequences for the jobs in the placement and claiming queues which would be lost, but not for the jobs submitted through KOALA that are already running. Of course, we can make the KOALA scheduler more reliable by periodically writing the contents of its queues to disk and by reading these contents after recovery. This is a feature worth considering when KOALA runs in a more unreliable heterogeneous environment and not (only) on the DAS system.

The reliability of the runners depends on their implementations. For example, the DRunner is less reliable than the KRunner and the IRunner because of bugs in the MPICH-G2 and the DUROC libraries. The DUROC library has not been updated since Globus Toolkit version 2.4, and the last update of MPICH-G2 was made at the end of 2005. The DRunner copes with these unreliability issues by simply restarting failed jobs. It is possible from time to time for a runner instance to crash. We should point out that due to our modular design, a crash of any instance of a runner does not affect the operation of the scheduler, nor does it affect any other job than the job to which the runner belongs. When a runner crashes, we want all the processors it was using to be immediately made available for other jobs. To do so, the runners framework traps all the interrupts signaled by the operating system and performs cleanup accordingly, before exiting. The scheduler also monitors all the runners by periodically sending them are-you-alive messages. A runner that does not respond to such a message is assumed to have crashed, and the processors allocated to the corresponding job are immediately made available again.

## 3.8 Experiences with Globus

The DAS testbed uses the Globus Toolkit as its grid middleware, which means that KOALA on the DAS relies on Globus for grid middleware services. KOALA has used Globus Toolkit version 2.4 and version 3.02 on DAS-2, and at the time of writing this thesis on DAS-3, KOALA uses version 4.04 of Globus. Globus has existed in two flavors since its version 3, pre-Web Services (pre-WS) and Web Services (WS), with the pre-WS flavor used by KOALA. Studying web services and porting KOALA to web services is beyond the scope of this thesis.

Initially, the KOALA scheduler used the Metacomputing Directory Service (MDS) and

the Replica Location Service (RLS) of the Globus Toolkit in its Information Service component. However, the MDS delayed greatly the operation of the scheduler by delaying the dynamic information about node status, and sometimes, no updated information about node status was received at all. Puppin et al. [82] and Aloisio et al. [29] discuss in great detail the shortcomings of the Globus MDS. As mentioned in Section 3.1.1, KOALA on the DAS uses native mechanisms to obtain status information of nodes, and no plans to test the MDS that comes with Globus Toolkit 4 have been made. The shortcomings we encountered with the Globus RLS were due to its use of the Lightweight Directory Access Protocol (LDAP); on the DAS, the permissions to add or modify records on the centrally managed LDAP server were granted only to privileged users. This contradicted with our design goal of allowing any user the ability to add or modify their records, hence the reason to maintain our own database to provide the mappings of the logical names of files to their physical locations.

The KOALA runners framework and the runners are the active users of Globus on the DAS. The authentication provided by the Globus Security Infrastructure (GSI) and the Globus Specification Language (RSL) are used extensively by the runners, and so far, no problem have been reported with these two Globus components. The current runners also use the Globus Resource Allocation Manager (GRAM) and both Globus gridFTP and Global Access to Secondary Storage (GASS), to launch jobs for execution on remote sites and to transfer files, respectively. The limitations of the pre-WS GRAM are discussed clearly by Dumitrescu et al. [56]. During our tests and experiments, the performance degradation of GRAM discussed in [56] were felt when the number of concurrent jobs submitted to the same cluster reaches a certain threshold. We discuss more about the overhead of the GRAM in the experiments in Chapter 6. The limitations of gridFTP that were observed during our tests were all due to the well-known issues and limitations of the gridFTP server. The most prominent issue was the random hanging of transfers after they have finished, which occurred when the number of concurrent transfers was high and when running a striped transfer with a parallelism of more than 1.

In general, our experience with using pre-WS Globus with the runners framework and the runners has been positive; however, a comparative assessment of Globus can only be made when more experience with running KOALA on other middleware has been gained, which is beyond the scope of this thesis.

## 3.9 Experiences with KOALA

KOALA has been operational in the DAS testbed since September 2005. So far, more than 500,000 jobs have been submitted successfully with KOALA, both for testing KOALA and for doing useful work with it. In this section we describe four examples from the wide range of usage of KOALA in the past two years.

Philips Research in Eindhoven in the Netherlands [20] has designed a Grid Architecture for Medical Applications (GAMA) that enables several relevant compute-intensive medical applications to use grid technologies, for improved performance and cost-effective access to large numbers of various resources [41]. In the GAMA architecture, there is a dedicated Grid Access Point (GAP), which sends requests from client(s) through a Windows-based interface in hospitals to the grid and returns results. During the research for and development of their medical applications, DAS-2 was used to provide compute resources as provisioned by KOALA. In the setup, the DRunner was installed in the GAP and was used to execute non-fixed jobs on the DAS-2 clusters selected by the KOALA scheduler. This setup proved to be successful and it allowed researchers in Philips to only concentrate on grid application development.

The SURFnet Gigaport project investigated the feasibility of the grid for running network tests with GridFTP. One of the objectives of this project was to see if the results of the network tests can be used to inform users and maintainers of the grid about possible network problems and to influence their choice of grid resources. To realize this objective, a periodic performance monitor was run, first between two nodes at a single cluster with a special 1 Gbit/s test network, then between any two nodes at any DAS-2 clusters. The selection of these nodes and the submission of the performance monitor was done by KOALA. KOALA was used because of its ability to select non-faulty idle nodes due to its mechanisms for fault tolerance, remote submission, and job monitoring, and due to its ease of use. The performance monitor was run periodically between November 2005 and May 2007, when the DAS-2 cluster of the University of Amsterdam was decommissioned. During this period, this performance monitor helped us to unearth some of the "hidden" bugs and making KOALA reliable. By the way, not only KOALA bugs were unearthed by this run but also some DAS-2 specific node specific errors (soft errors), which by solving them we managed to make the DAS multicluster system more reliable. The results of these network performance tests have been published on the website of [8].

In peer-to-peer (p2p) file sharing networks such as BitTorrent [2], peers that are in the process of downloading the same file have to be discovered, something that is called swarm discovery [88], before content can be shared with these peers. Roozenburg [88] proposes a decentralized swarm discovery protocol called LITTLE BIRD supported by Tribler [81], a social community application that facilitates file sharing through a p2p network. For the evaluation of LITTLE BIRD, an experimental environment called CROWDED, which enabled large-scale trace-based emulations of swarms on the DAS-2, was created. To emulate a large swarms in the DAS-2, the CROWDED environment used the KRunner to request nodes on different DAS-2 clusters from the KOALA scheduler, and to submit for execution the actual Tribler application to the allocated nodes. The performance of KOALA and the DAS during experiments with CROWDED were very good, as a result, KOALA will be used more extensively for the research in the Tribler project.

The possibilities of connecting a grid application to a visualization component have been investigated by van Ameijden [30]. In this study, a molecular dynamics simulation package called Gromacs [9] is used as the grid application and the MolDRIVE visualization package [16] as the visualization component. Both Gromacs and MolDRIVE are adapted to enable running molecular dynamics simulations in a distributed fashion on the DAS-2, with the simulations being manipulated from and their results being visualized on a Virtual Workbench. In this study, the KOALA DRunner was used to submit job components to see how the performance of the application is influenced by running the simulations on multiple sites, and whether the communication delays between the simulation and the visualization components of the application differ much across clusters.

# Chapter 4

# The KOALA Job Policies

In KOALA, there are two types of job policies, *placement policies* for placing the components of a job and *claiming policies* for claiming the processors previously allocated to the components. The placement policies are used to decide where the jobs should be sent for execution and the claiming policies have the task of ensuring that jobs are launched for execution at their planned time. It should be noted that new policies can be added, and that existing policies can be modified at any time without affecting the operation of the scheduler. In particular, two new placement policies called Cluster Minimization (CM) and Flexible Cluster Minimization (FCM) [91], which are communication-aware placement policies for non-fixed and flexible jobs, respectively, have been added during the writing of this thesis. The KOALA job policies can both be used with jobs that require co-allocation and those which do not require co-allocation.

We begin this chapter by presenting the job submission timeline in Section 4.1. This section also defines some parameters that are used in the subsequent sections. Jobs waiting to be placed or claimed by any of the KOALA policies are held in one of the KOALA *placement queues* or in the *claiming queue*, respectively. Section 4.2 presents these KOALA queues. In Section 4.3 we discuss two placement policies, the Close-to-Files (CF) placement policy and the Worst-Fit (WF) placement policy. The CF policy has the goal of reducing the waiting times of jobs by minimizing their file transfer times. On the other hand, the WF policy simply tries to optimize the placement procedure by balancing the use of the grid resources (processors). A claiming policy called the Incremental Claiming Policy (ICP) that is used in the absence of processor reservation by local resource managers, is discussed in Section 4.4.

## 4.1 Job Submission Timeline

In Section 3.6, we have seen that a job moves from one phase to another in KOALA at stipulated times or after the success of some operations at certain times. For example,

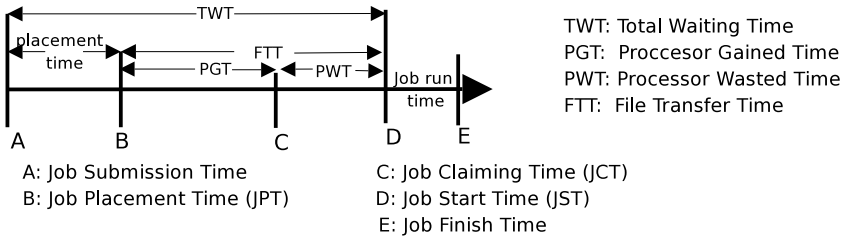



Figure 4.1: The timeline of a job submission.

after a successful placement of a job at a certain time, the job is moved to the claiming phase where the claiming for processors for its components will begin at a designated claiming time. These times together form the job submission timeline shown in Figure 4.1 that we present in this section.

A new job request is received by a runner at a time called the Job Submission Time, which is point A in Figure 4.1. This is the time that a user has launched a specific runner for his/her job. The time when the placement of the job succeeds, i.e., the successful placement of its last component, is called the Job Placement Time (JPT), which is point B in Figure 4.1. After the job has been forwarded for claiming, the time when claiming processors for the job components starts is called the Job Claiming Time (JCT), point C in Figure 4.1. Point D is the time when the job should be launched, which is the so-called Job Start Time (JST). The JST is estimated as the sum of job's JPT and its File Transfer Time (FTT), which is calculated as the maximum of the estimates of the file transfer times of all of its components. The time from the submission of the job until its actual launch time is called the Total Waiting Time (TWT) of the job. The difference between JCT and JPT, and JST and JCT, which are referred to as the Processor Gained Time (PGT) and the Processor Wasted Time (PWT), are discussed in Section 4.2.2. The job finishes execution at point E, which is called the Job Finish Time.

It is possible for a job to fail after its initial successful placement and be restarted. If a job is restarted a number of times, the values for JPT, JCT, and JST will be the last values recorded, which corresponds to the placement that leads to the successful execution of the job. The value for Job Submission Time does not change, however.

## 4.2 The KOALA Queues

KOALA maintains placement queues and a claiming queue to hold jobs that currently cannot be placed or for which processors for their components currently cannot be claimed, respectively. The placement and/or the claiming procedure for a job may fail due to the unavailability of enough idle processors as requested by a job at its execution sites. The placement and claiming procedures for jobs in the queues are retried for

a fixed number of times and if then still unsuccessful, the job fails and is deleted from the queues. Sections 4.2.1 and 4.2.2 discuss the KOALA placement queues and claiming queue, respectively.

### 4.2.1 The placement queues

A new job arriving to KOALA is appended to the tail of the KOALA placement queue corresponding to its priority. KOALA maintains four placement queues, one for each of the priorities. These queues are: the super-low placement queue for super-low priority jobs, the low placement queue for low priority jobs, the high placement queue for high priority jobs, and the super-high placement queue for super-high priority jobs. These queues hold all jobs that have not yet been placed. KOALA regularly scans the placement queues according to their priorities from head to tail to see whether any job in them can be placed. This means that jobs in same placement queue are considered for placement in their arriving order (FCFS) during each scan. If the first job cannot be placed, in effect a backfilling approach [73] is attempted whereby jobs further down the queue are possibly placed without taking into account any placement delay to the first job.

KOALA selects a queue to scan based on its priority in a round robin manner. To give jobs of higher priorities more chance to be placed we assign a weight to each queue, which determines the number of times that queue will be scanned before the next queue. Below we present our technique for queue selection.

When performing queue selection, KOALA first groups the super-high and high placement queues to form the *higher placement queues*, and the low and super-low placement queues to form the *lower placement queues* as shown in Figure 4.2. The higher placement queues are scanned first $N_h$ times before scanning the lower placement queues $N_l$ times, with $N_h \geq N_l \geq 1$. In each scan of the higher placement queues, the super-high placement queue is scanned $n_1$ times before scanning the high placement queue $n_2$ times, where $n_1 \geq n_2 \geq 1$. This means that after $N_h(n_1 + n_2)$ scans we begin scanning the lower placement queues. Likewise, in each scan of the lower placement queues, the low placement queue is scanned $n_3$ times before scanning the super-low placement queue $n_4$ times, where $n_3 \geq n_4 \geq 1$. This also means we scan again the higher placement queues after $N_l(n_3 + n_4)$ scans of the lower placement queues. As an example, if $N_h$, $N_l$, and the $n_i$, $i = 1, \ldots, 4$ are all set to 1, then the queues are selected in a traditional round robin manner where all queues are of equal priority. On the other hand, if $N_1$ is set 2 and the rest of the weights are set to 1, this means we scan the higher placement queues twice before we scan the lower placement queues once.

This queue selection technique shares the same idea as the technique called Group Ratio Round-Robin presented by Caprita et al. [45] where groups of clients are selected in a round robin manner based on the ratio of their group weights.

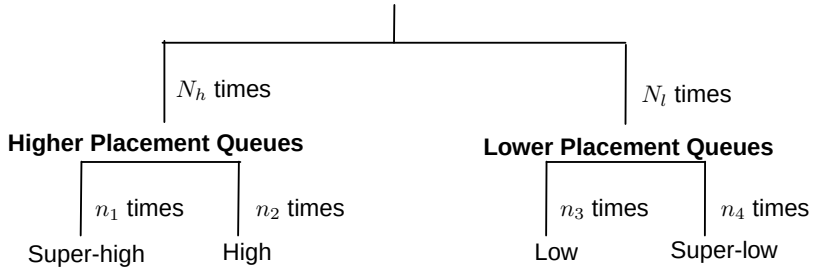



Figure 4.2: Grouping of priority levels and the number of times KOALA scans the corresponding placement queues.

The time between successive scans of the placement queues is a fixed interval (this time, $N_h$, $N_l$, and the $n_i$, $i = 1, \ldots, 4$, are parameters of KOALA); the placement queues which are empty, are simply skipped. The time when the placement of a job succeeds is called its JPT, depicted in Figure 4.1, which shows the timeline of a job submission. The figure also shows the *placement time*, which is the difference between JPT and the time the job enters the placement queue.

For each job in a placement queue we maintain its number of *placement tries*, i.e., the number of scans of the queue while it contains the job. When this number exceeds a threshold, the job submission fails. This threshold can be set to $\infty$, i.e., no job placement fails.

With our current setup, starvation is possible for any job in the KOALA placement queues due to a high load of higher-priority jobs or local jobs. To minimize starvation as much as possible in KOALA, jobs in the low placement queue or in the high placement queue move one priority level up after every $P$ placement tries until they reach the super-high placement queue; $P$ is another KOALA parameter, which can be set to $\infty$ to prevent jobs from changing their priorities. Jobs in the super-low placement queue are not allowed to change their priority level.

The KOALA parameters presented in section need to be determined before running the KOALA scheduler. Currently, we have assigned default values to these parameters based on the observations made after a number of test runs. Of course, these values can be fine tuned depending on the multicluster system where the scheduler runs and the types of jobs submitted to KOALA. Of all the parameters, KOALA is most sensitive to the interval between successive scans of the placement queues. The value of this parameter is a trade-off between too much CPU load due to excessive scans and too long waiting times of jobs in the placement queues.

### 4.2.2 The claiming queue

After the successful placement of a job, its File Transfer Time (FTT) and its Job Start Time (JST) are estimated before the job is added to the claiming queue. This queue holds jobs that have been placed and currently are waiting to be submitted for execution. The job's FTT is calculated as the maximum of all of its components' estimated file transfer times (see Section 4.3.1 for how we compute these), and the JST is estimated as the sum of its JPT and its FTT as shown in Figure 4.1. Our challenge here is to guarantee processor availability at the JST. In the absence of processor reservation in the LRMSs, the KOALA scheduler can immediately claim processors for a job at its JPT and allow the job to hold the processors until its JST. However, this is wasteful of processor time. It is also possible for KOALA to claim processors only at JST but then there is the risk of processors not being available anymore. Therefore, to minimize the Processor Wasted Time (PWT), which is the time the processors are held but not used for useful work, and at the same time increase the chance of claiming success, an attempt to claim processors for a job is done at the job's so-called Job Claiming Time (JCT) (point C in Figure 4.1). A job's JCT is initially set to the sum of its JPT and the product of $L$ and FTT:

$$JCT_0 = JPT + L \cdot FTT,$$

where $L$, which is a real number between 0 and 1, is a parameter assigned to each job by KOALA. The initial value of $L$ assigned to jobs by KOALA is decided by an administrator, e.g., 0.75, and this value is updated dynamically during the claiming attempts. It should be noted that if 0 is assigned to $L$ then the claiming procedure will be attempted at JPT, and if 1 is assigned to $L$ then the claiming procedure will be attempted at JST. More on how $L$ is updated is described below. In the claiming queue, jobs are arranged in increasing order of their JCT.

KOALA tries to claim for a job (*claiming try*) at the current JCT by using our Incremental Claiming Policy, which is described in Section 4.4. Claiming for a component at the current job claiming try succeeds if all processors it has requested can be claimed, otherwise claiming fails. The success of claiming for all components of the job results in the success of the claiming try. The job is removed from the claiming queue if the claiming try is successful. Otherwise, we perform successive claiming tries. For each such try we recalculate a new JCT by adding to the current JCT the product of $L$ and the time remaining until the JST (time between points C and D in Figure 4.1):

$$JCT_{n+1} = JCT_n + L \cdot (JST - JCT_n).$$

If the job's $JCT_{n+1}$ reaches its JST and still claiming for some components fails, the job is returned to the placement queue. Before doing so, its parameter $L$ is decreased by a fixed fraction, e.g., 0.25, and its components that were successfully started in the previous

claiming tries are aborted. The parameter $L$ is decreased each time the JST is reached until it hits its lower bound, e.g., 0, so as to increase the chance of success of claiming. If the number of claiming tries for a job exceeds some threshold (which can be set to $\infty$), the job submission fails.

For a job, a new JST is estimated each time the job is returned to the placement queue and re-placed with a placement policy. We define the *start delay* of a job to be the difference between the final JST where the job execution succeeds and the original JST. It should be noted that the re-placements result in multiple placement times (the placement time has been defined in Section 4.2.1). Therefore, we define the *total placement time* of a job as the sum of all its placement times.

As we saw in Section 4.1, we call the time between the JPT of a job and the time of successfully claiming processors for it, the Processor Gained Time (PGT) of the job. The PGT is depicted in Figure 4.1. During the PGT, jobs submitted through other schedulers than our grid scheduler can use the processors.

## 4.3 The Job Placement Policies

The KOALA scheduler uses job placement policies to select the execution sites with enough idle processors for the components of non-fixed, semi-fixed, and flexible jobs. If the job components require input files, the scheduler also uses the placement policies to select the file sites such that the time to transfer the input files to the selected execution sites is minimal. The placement policies presented in this section support co-allocation by employing *atomic placement*, i.e., the placement of a co-allocated job is only successful if all of its components can be placed at one time. The placement policies have no restrictions on how to distribute the components of a co-allocated job across the sites of a grid, and there is a chance that more than one or even all components of a job are placed on a single execution site. In the absence of the use of co-allocation, the Close-to-Files policy defined below can be used to place single-component jobs close to a file site of its input file, and the Worst-Fit placement policy defined below can be used to balance the load across the grid.

In this section we present two of the placement policies operational in the KOALA scheduler. These policies are the Close-to-Files placement policy discussed in Section 4.3.1, and the Worst-Fit placement policy discussed in Section 4.3.2.

### 4.3.1 The Close-to-Files placement policy

Placing a non-fixed job in a multicluster means finding a suitable set of execution sites for all of its components and suitable file sites for the input file. (Different components may get the input file from different locations.) The most important consideration here is of

course finding execution sites with enough processors. However, when there is a choice among execution sites for a job component, we choose the site such that the (estimated) delay of transferring the input file to that site is minimal. We call the placement algorithm doing just this the Close-to-Files (CF) policy [78]. It uses the following parameters in its decisions:

- **The numbers of idle processors in the sites of a grid:** A job component can only be placed on an execution site which will have enough idle processors at the job start time.
- **The file size:** The size of the input file, which enters in the estimates of the file transfer times.
- **The network bandwidths:** The bandwidth between a file site and an execution site gives the opportunity to estimate the transfer time of a file given its size.

**Algorithm 1** Pseudo-code of the Close-to-Files job-placement algorithm.

```
 1: order job components according to decreasing size
 2: for all job component j do
 3:    S_j ⇐ set of potential execution sites
 4:    if S_j ≠ ∅ then
 5:       select an E ∈ S_j
 6:    else
 7:       P_j ⇐ set of potential pairs of execution site, file site
 8:       if P_j ≠ ∅ then
 9:          for all (E, F) ∈ P_j do
10:             estimate the file transfer time T_(E,F)
11:          select the pair (E, F) ∈ P_j with minimal T_(E,F)
12:          for all file site F' of the job do
13:             insert (E, F') into the history table H
14:       else
15:          job placement fails
```

When given a job to place, CF operates as follows (the line numbers mentioned below refer to Algorithm 1). CF first orders the components of a job according to decreasing size (line 1), and then tries to place the job components in that order (loop starting on line 2). The decreasing order is used to increase the chance of success for large components.

For a single job component $j$, CF first determines the set $S_j$ of *potential execution sites* (line 3); these are the file sites of the job that have enough idle processors to accommodate the job component. If $S_j$ is not empty, CF picks an element from it as the execution site

of the component (line 5). (We currently have a function that returns the names of the file sites in alphabetical order, and CF picks the first.)

If the set $S_j$ of potential execution sites is empty (line 6), we might consider all pairs of execution sites with sufficient idle processors and files sites of the job, and try to find the pair with the minimal file transfer time. This is not efficient in large grids with many sites; therefore, CF maintains a *history table* $H$ with a subset of pairs of execution sites and file sites to consider. From $H$, CF selects all *potential pairs* $(E, F)$ of execution site, file site, with $E$ having a sufficient number of idle processors for the job component and $F$ being a file site of the job (line 7). If no such pair exists in $H$, the job component, and in case of co-allocation the whole job, currently cannot be placed (line 15). Otherwise, CF estimates for each selected pair the file transfer time from the file site to the execution site (line 10), and picks the pair with the lowest estimate (line 11). If $(E, F)$ is the pair selected, CF inserts into $H$ all pairs $(E, F')$ with $F'$ a file site of the job (lines 12, 13). Note that if the history table is initially empty, it will remain empty. Therefore, it has to be initialized with some set of suitable pairs of execution and file sites.

### 4.3.2 The Worst-Fit placement policy

Built into KOALA is also the Worst Fit (WF) placement policy. WF places the job components in decreasing order of their sizes on the execution sites with the largest (remaining) numbers of idle processors. The decreasing order is used to increase the chance of success for large components. In case the files are replicated, WF selects for each component the replica with the minimum estimated file transfer time to that component's execution site. Like with CF, placement of a job fails if the placement of any of its components fails when co-allocation is used with WF. As mentioned in the introduction of this section, both CF and WF make perfect sense in the absence of co-allocation, where WF in particular balances the load well across the multicluster system.

## 4.4 The Incremental Processor Claiming Policy

Jobs submitted with KOALA share processors with jobs of local cluster users. When claiming processors by KOALA, it is possible for processors previously allocated by a placement policy to be used by local jobs due to the absence of reservation mechanisms in LRMSs. It is also possible that at the time of claiming processors, they are marked unusable by an LRMS due to errors. In KOALA, claiming processors for a job starts at a job's initial JCT, and if not successful, is repeated at subsequent claiming tries. For components for which claiming has failed, it is possible to increase their chance of claiming success in subsequent claiming tries by finding other sites with enough idle processors to execute them, or by preempting jobs of lower priorities to make processors available.

We call the policy doing exactly this the Incremental Claiming Policy (ICP), which operates as follows (the line numbers mentioned below refer to Algorithm 2). For a job, ICP first determines the sets $C_{prev}, C_{now}$, and $C_{not}$ of components that have been previously started, of components that can be started now based on the current numbers of idle processors, and of components that cannot be started based on these numbers, respectively. It further calculates $F$, which is the sum of the fractions of the job components that have previously been started and of the components that can be started in the current claiming try (line 1). We define $T$ as the required lower bound of $F$; the job is returned to the claiming queue if its $F$ is lower than $T$ (line 2).

**Algorithm 2** Pseudo-code of the Incremental Claiming Policy

**Require:** set $C_{prev}$ of previously started components of $J$ (initially $C_{prev} = \emptyset$)
**Require:** set $C_{now}$ of components of $J$ that can be started now
**Require:** set $C_{not}$ of components of $J$ that cannot be started now
1: $F \Leftarrow (|C_{prev}| + |C_{now}|)/|J|$
2: **if** $F \geq T$ **then**
3:  **if** $C_{not} \neq \emptyset$ **then**
4:  **for all** $j \in C_{not}$ **do**
5:  $(E_j, F_j, ftt_j) \Leftarrow Place(j)$
6:  **if** $JCT + ftt_j < JST$ **then**
7:  $C_{now} \Leftarrow C_{now} \cup \{j\}$
8:  **else if** $priority(j) \neq$ super-low **then**
9:  $P_j \Leftarrow count(processors)$ `/* used by jobs of lower priorities than` $j$ `and idle at` $E_j$ `*/`
10:  **if** $P_j \geq$ size of $j$ **then**
11:  **repeat**
12:  Preempt lower priority jobs at $E_j$
13:  **until** $count(idle\ processors) \geq$ size of $j$ `/* at` $E_j$ `*/`
14:  $C_{now} \Leftarrow C_{now} \cup \{j\}$
15:  start components in $C_{now}$

For each component $j$ that cannot be started on the cluster selected when placing the job, ICP first tries to find a new pair of execution site-file site with the placement policy originally used to place the job (line 5). On success, the new execution site $E_j$, file site $F_j$, and the new estimated transfer time between them, $ftt_j$, are returned. If it is possible to transfer the file between these sites before JST (line 6), the component $j$ is moved from the set $C_{not}$ to the set $C_{now}$ (line 7).

For a job of priority other than super-low, if the re-placement of the component fails or the file cannot be transferred before JST (line 8), ICP performs the following. At the execution site $E_j$ of component $j$, it checks whether the sum of the number of idle processors and the numbers of processors currently being used by jobs of lower priorities

is at least equal to the number of processors the component requests (lines 9 and 10). If so, the policy preempts lower priority jobs in descending order of their JST (newest-job-first) until a sufficient number of processors have been freed (lines 11-13). The preempted jobs are then returned to the placement queues.

Finally, those components for which processors can be claimed at this claiming try are started (line 15). Synchronization of the start of the components at the JST depends on the application type and therefore, it is specific to each runner. For example, with the DRunner, synchronization is achieved by making each component wait on the job barrier until it hears from all the other components.

When $T$ is set to 1, the claiming process becomes *atomic*, i.e., claiming only succeeds if for all the job components processors can be claimed.

# Chapter 5

# Evaluation of the KOALA Scheduler

The KOALA scheduler comprises the placement and claiming queues, and the job policies, which are used to manage the jobs in the queues. The job policies, which have been presented in Chapter 4, have been designed to address specific issues of the grid infrastructure. The Close-to-Files (CF) placement policy addresses the problem of long delays when starting a job because of long input file transfers, and the Worst-Fit (WF) placement policy balances the number of idle processors among the clusters while trying to minimize file transfer times, too. The Incremental Claiming Policy (ICP) on the other hand, tries to make processors available for job components, if necessary by finding processors at other sites than selected by a placement policy or, if permitted, by forcing processor availability through the preemption of running jobs. All these policies have been designed to enable co-allocation whenever possible. The performance evaluation of the KOALA job policies is the subject of Section 5.1.

The KOALA scheduler needs to deal with the dynamicity of the grid resources and with reliability problems of grid components to ensure that grid jobs are completed successfully. In November 2004, a major upgrade of the operating system (from RedHat version 7.2 to RedHat Enterprise Linux version 3) was done in all DAS-2 clusters. In addition to the operating system upgrade, the clusters' local resource manager, openPBS, was also upgraded. Upgrading the operating system prompted some of the important system libraries like the Myrinet binaries, which are crucial for communication within a single cluster, to be rebuilt. Right after this major upgrade, the DAS testbed was very unreliable and running meaningful experiments was difficult. However, the unreliability of the DAS during this period provided us with a chance of testing KOALA in a grid-like environment where the job failure rate is high. In Section 5.2 we present the results of the experiments with KOALA performed while the DAS testbed was unreliable. All experiments reported in this chapter were done in the DAS-2 testbed.

Extensive simulation studies of processor co-allocation in multicluster systems have been presented in a number of publications. In Section 5.3 we discuss as to what extent

the results of these simulations and of our experiments can be compared.

## 5.1 Evaluation of the KOALA Job Policies

In this section we present the experiments we have conducted to assess the performance of our placement and claiming policies. In the experiments we assess the performance of these policies on a stable DAS system and when the system was saturated with many jobs. One of the important things we assess is the level of co-allocation actually used by the policies when placing jobs. During the experiments, we did not have control over jobs from local users, which form the so-called *background load*. In these experiments we also try to assess the impact of a high background load on KOALA. Sections 5.1.1–5.1.3 present the setup of the experiments, while Sections 5.1.4–5.1.8 discuss the results of these experiments.

### 5.1.1 KOALA setup

In the experiments in this section, KOALA is setup as follows. No limits are imposed on the number of job placement and claiming tries to avoid forced job failures when the limits are reached. All jobs have the same priority level and therefore, only one placement queue is used. The interval between successive scans of the placement queue is fixed at 1 minute, which as observed in KOALA logs, is a trade-off between too much CPU load due to excessive scans and too long waiting times of jobs in the placement queue. From the KOALA logs, it is also observed that for most jobs, claiming is successful at a value of parameter $L$ determining when to start claiming between 0.5 and 0.75. Hence, the initial value of the $L$ is set at 0.75. As there are only five clusters in our testbed, we initialize the history table $H$ to contain all possible pairs of execution sites and file sites. The parameter $T$ of our claiming algorithm, described in Section 4.4, is set to 1, so claiming is atomic. The KRunner was used to submit jobs in these experiments because it was the only runner available. Using the KRunner makes perfect sense since we want to assess the job policies only at the scheduler level.

### 5.1.2 The workload

We put two workloads of jobs to be co-allocated on the DAS-2 that all run the Poisson application, which has been described in Section 2.3.2, in addition to the regular workload of the ordinary users. In the workloads, a high priority is assigned to all jobs to give them equal placement opportunities. Both workloads have 200 jobs, with the first workload $W_{30}$ utilizing on average 30% of the whole system and the second, $W_{50}$, utilizing on average 50% of the DAS-2. In the workloads, jobs have 1, 2, or 4 components requesting

the same number of processors, which can be 8 or 16. So the total size of the jobs ranges from 8 to 64. All possibilities for combining the number of components and the number of processors have equal probability. Each job component requires the same single file of either 2, 4, or 6 GByte. Again, all possibilities for combining the size of a job and the file size have equal probability. For the input files we consider two cases, one without file replication and another with each file randomly replicated in three different sites. Since the jobs in our workloads were submitted with the KRunner, the version of the Poisson application which is not grid-enabled is used, which means that the components of a single job are executed independently of each other. The average execution time of this application with components of size 8 and 16 is 192.0 and 90.0 seconds, respectively. At the start of the experiments, the numbers of processors in the DAS clusters were as shown in Table 5.1. We assume the arrival process of our jobs at the submission site to be Poisson, where the arrival rate has been calculated using the above parameters so that 30% or 50% of the system is utilized. Based on the arrival rate of the jobs, with workload $W_{30}$ the last job arrived at around 6500 seconds after the arrival of the first job, and with the $W_{50}$, the last job arrived after 3900 seconds.

Table 5.1: The numbers of processors in the DAS-2 clusters at the start of the experiments reported in Section 5.1.

| Cluster Location | Number of Processors |
|---|---|
| Vrije University | 144 |
| Leiden University | 56 |
| University of Amsterdam | 56 |
| Delft University | 64 |
| Utrecht University | 64 |

### 5.1.3 Background load

One of the problems we have to deal with is that we do not have control over the background load imposed on the DAS by other users. These users submit their (non-grid) jobs straight to the local resource managers, bypassing KOALA. During the experiments, we monitor this background load and we try to maintain it at 30% in each cluster. Since maintaining the background load exactly at 30% is impractical due to the dynamicity of jobs of local users, we allow this value to increase up to 40%. When this utilization falls below 30% in some cluster, we inject dummy jobs just to keep the processors busy. When this utilization rises above 40% with some of this utilization contributed by our dummy jobs, we kill the dummy jobs to lower the utilization to the required range. Our experimental conditions are no longer satisfied if during the experiments, the background load

in a cluster rises above 40% and stays there for more than a minute. In such a case, the experiment were aborted and repeated.

### 5.1.4 Presentation of the results

We will present the results of each experiment in this section with several graphs. The first of these shows the different utilizations in the system. Here, the background load is the utilization due to the jobs of the regular DAS-2 users. The KOALA load is the utilization due to the co-allocation workload described in Section 5.1.2 *without* the load incurred between the Job Claiming Time (JCT) and the Job Start Time (JST). We also show the Processor Wasted Time (PWT) utilization, which is the fraction of the system capacity wasted because KOALA starts jobs earlier than the JST, and the Processor Gained Time (PGT) utilization, which is the fraction of the system capacity gained because KOALA does not claim processors immediately when job placement is successful. As KOALA keeps track of the processors on which it has placed jobs and does not also allocate these processors to other jobs, the PGT utilization can only be used by local jobs (or jobs submitted by other grid schedulers), but not by other jobs submitted through KOALA.

The second graph shows the average Job Placement Time (JPT) and the File Transfer Time (FTT) (the sum of these two is the Total Waiting Time (TWT)). Only for workload $W_{30}$ do we also present more detailed statistics than simply the average of the placement times. The final graph presents the average numbers of placement and claiming tries. In these two and in later graphs, we use the notation $C \times S$ to indicate jobs with $C$ components of size $S$. It should be noted here that the results for the different workloads are presented with different scales to make them better visible.

### 5.1.5 Results for the workload of 30%

Figure 5.1 shows the different utilizations in the system for the CF and WF policies for workload $W_{30}$. During these experiments, the total utilization is about 70%. Because the experiments finished shortly after the submission of the last job and the actual co-allocation load (KOALA load) is about 30% for both CF and WF with and without replication, we conclude that the system is stable with workload $W_{30}$. It is also shown in this figure that the utilization wasted while jobs wait for file transfers to complete (PWT) is about 2%. This percentage is very low compared to the utilization gained by postponing claiming (PGT), which ranges between 6% and 9%. This shows that our claiming mechanism, which is described in Section 4.2.2, works well in a stable system.

Figure 5.2 shows that the average job FTT with the CF policy is smaller than with the WF policy, both with and without replication. Furthermore, with replication, CF is more successful in finding execution sites “closer” to the file sites, which results in a smaller

average job FTT. As a result, the average TWT of the jobs is also reduced. The decrease in the average job FTT when the files are replicated for CF is expected because the number of potential execution sites as defined in Section 4.3.1 increases.

We also find in Figure 5.2 that for both placement policies, the average placement time increases as the number or the size of the job components increases, both with and without replication. The explanation for this is that more time is likely to be spent waiting for clusters to have enough processors available simultaneously. Also, as the number of components increases, the job FTT goes up because more files are likely to be moved.

In order to give more detail, In Figure 5.3 we show more statistics than only the average of the distribution of the placement times for each job size. In this figure we observe that the 25th percentile and the median of the placement times of the jobs are (very close to) zero. On the other hand, the 90th percentile is high for most job sizes. This means that the average placement times reported in Figure 5.2 are heavily influenced by few jobs with very high placement times.

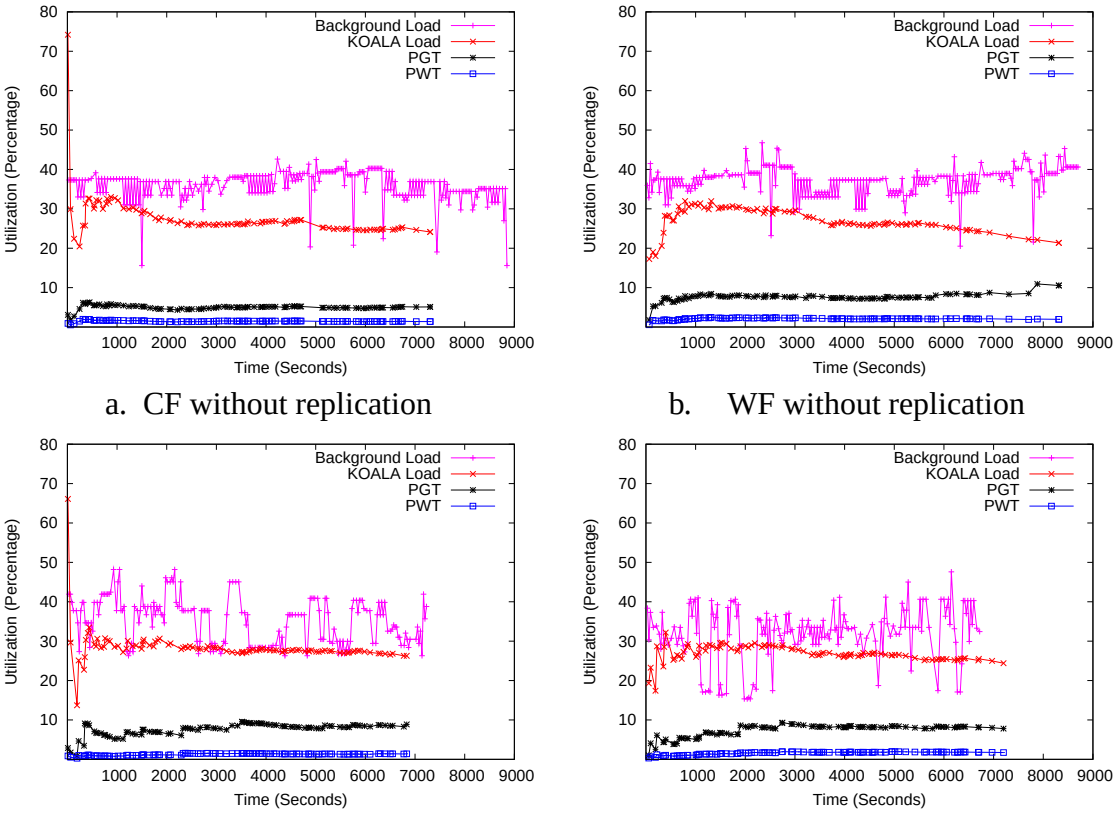
Figure 5.4 shows the average number of placement and of claiming tries for different job sizes. In the figure, the number of placement tries increases as the number or the size of the job components increases. The increase in the number of placement tries is caused by the waiting for clusters to have enough processors available simultaneously. This also contributes to the rise in the average placement time since KOALA waits for an interval of 1 minute between successive scans of the placement queue. The average numbers of claiming tries for different job sizes with this workload are quite low (around 1). Note that an average number of claiming tries equal or close to 1 means that we succeed in claiming an amount of time equal to 0.75 $\times$ FTT after successful placement according to the description of Section 4.2.2, and that the PGT utilization is three times the PWT utilization.

Overall, based on Figures 5.1, 5.2, and 5.4, we conclude that the combination of CF and replication performs best.

### 5.1.6 Results for the workload of 50%

Figure 5.5 shows the utilizations of our experiments with workload $W_{50}$ for CF and WF with and without replication. Our main purpose with this workload is to see to what utilization we can drive the system. From the figure we find that the total utilization during our experiments is between 70% and 80%. However, the actual co-allocation load is well below 40%, the experiments are only finished long after the last job arrival, and the length of the placement queue goes up to 30, which shows that the system is saturated. So we conclude that we can drive the total utilization not higher than what we achieve here.

With this workload and with the CF policy, clusters “close” to files will often be occupied, forcing more long file transfers. As a result, the average FTT for CF and WF are



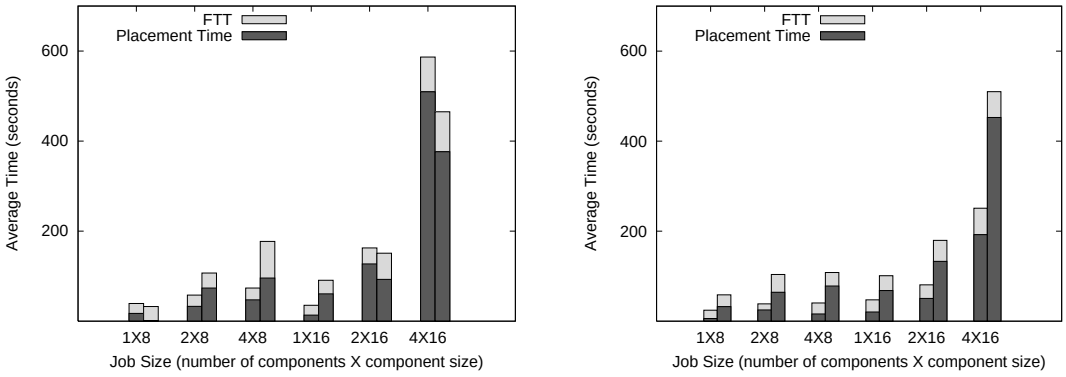

a. CF without replication

b. WF without replication

c. CF with replication

d. WF with replication

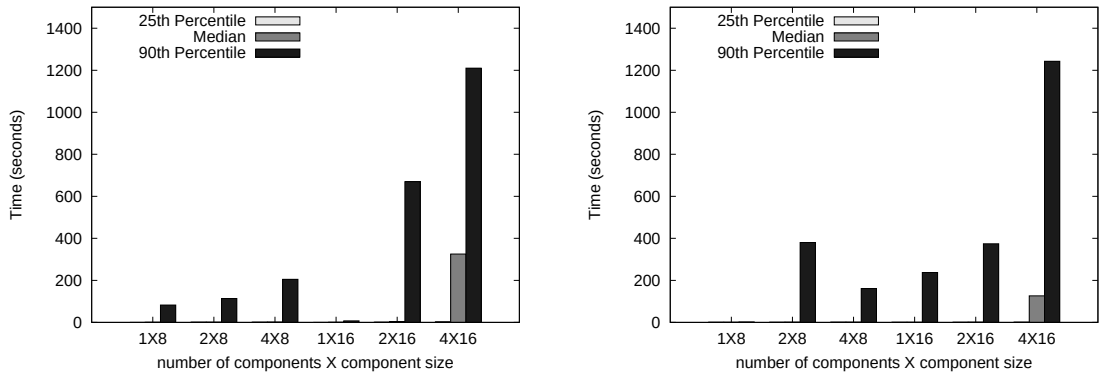
Figure 5.1: The utilizations for the Close-to-Files and Worst-Fit placement policies with workload $W_{30}$.




a. Without replication

b. With replication

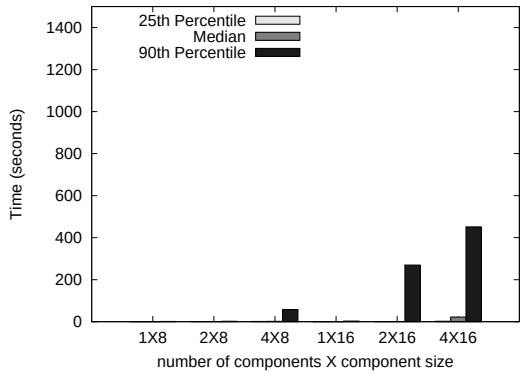
Figure 5.2: The average File Transfer Time and Placement Time for the Close-to-Files (left bars) and Worst-Fit (right bars) placement policies with workload $W_{30}$.



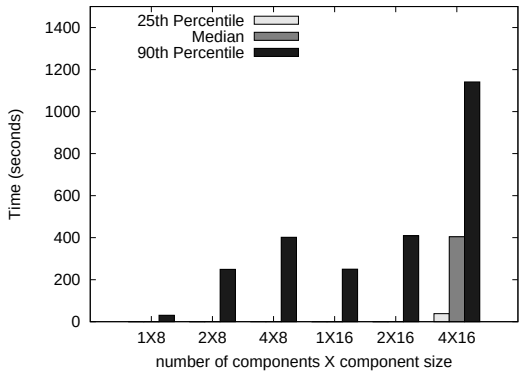

a. CF without replication




b. WF without replication

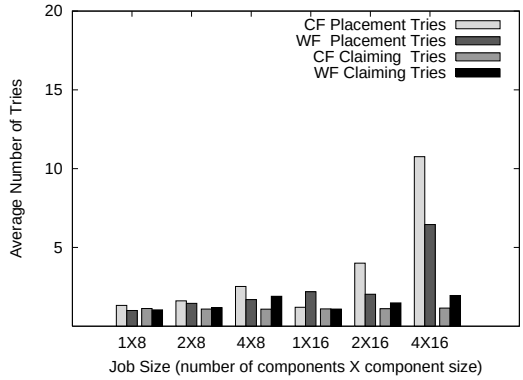



c. CF with replication

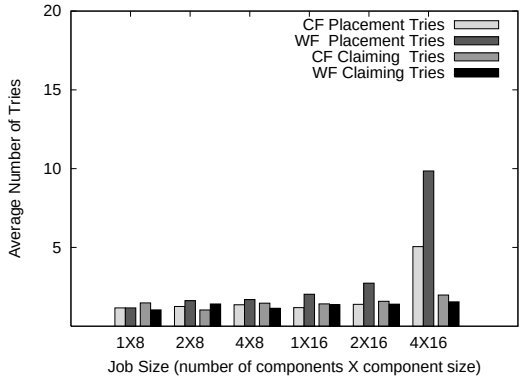



d. WF with replication

Figure 5.3: Statistics of the placement times for the Close-to-Files and the Worst-Fit placement policies with workload $W_{30}$ (all values are shown but the 25th percentile and median are (very close to) 0 in many cases).

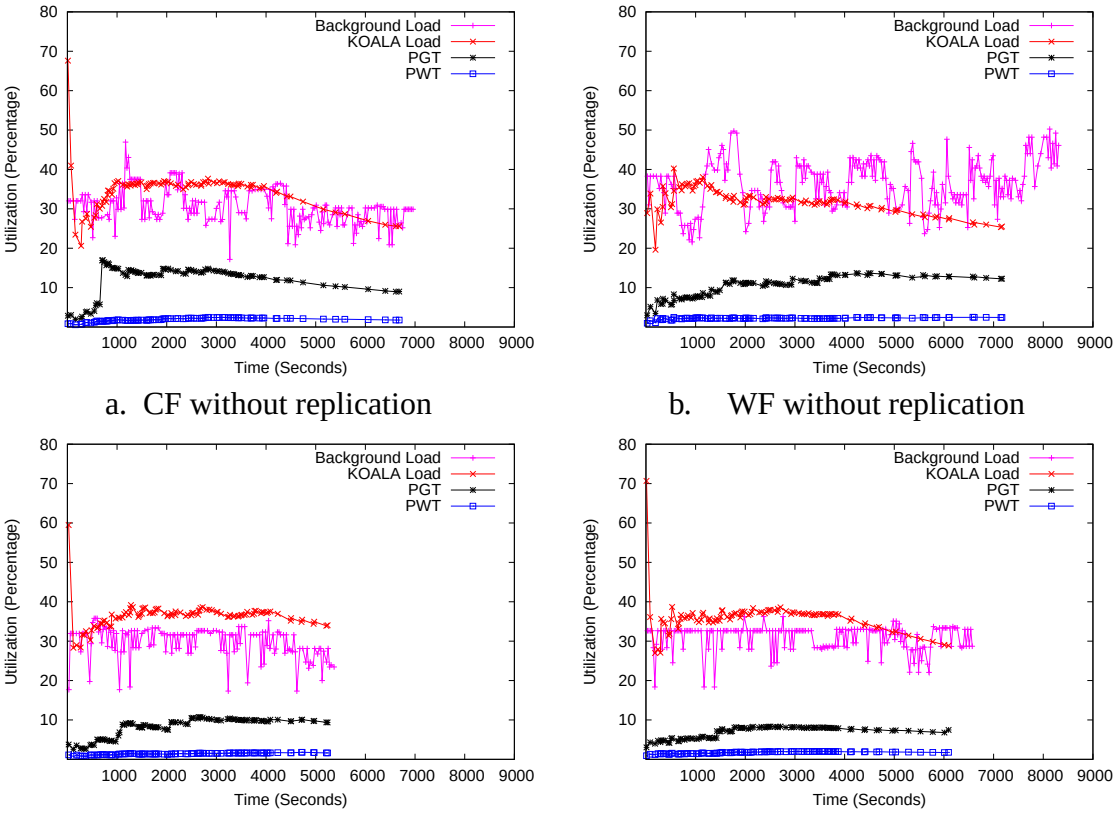



a. Without replication




b. With replication

Figure 5.4: The average number of placement and claiming tries with workload $W_{30}$.



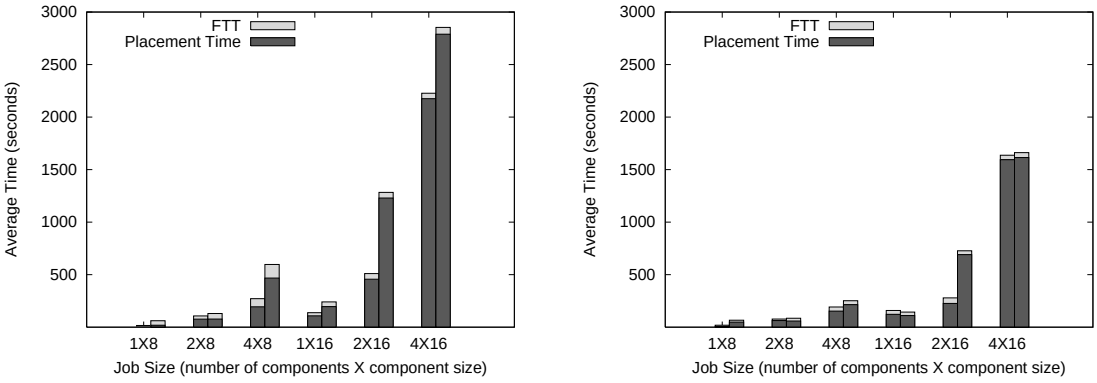

a. CF without replication

b. WF without replication

c. CF with replication

d. WF with replication

Figure 5.5: The utilizations for the Close-to-Files and Worst-Fit placement policies with workload $W_{50}$.

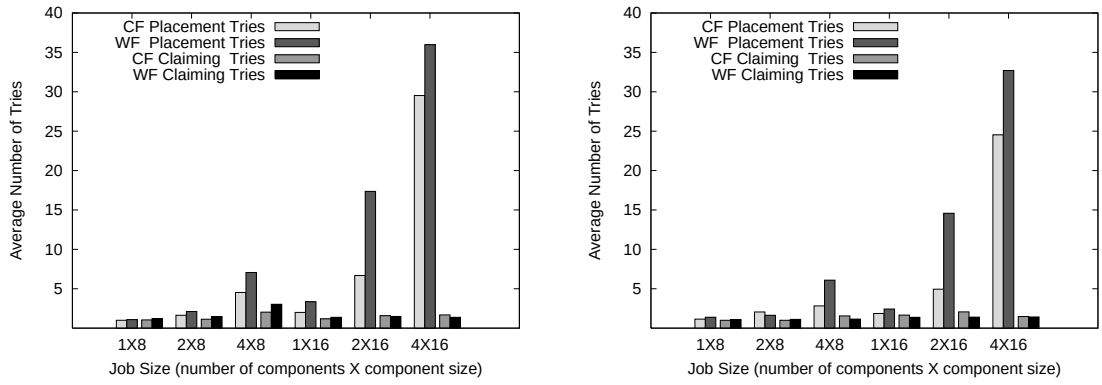



a. Without replication

b. With replication

Figure 5.6: The average File Transfer Time and Placement Time for the Close-to-Files (left bars) and Worst-Fit (right bars) placement policies with workload $W_{50}$.

relatively close to each other both with and without replication as shown in Figure 5.6 (note the different scale from Figure 5.2). Since the system is saturated with this workload, very much time is spent waiting for clusters to have enough processors available simultaneously. This explains the increase in the average placement times in Figure 5.6 and in the number of placement tries in Figure 5.7 as the number or size of the job components increases. However, similarly as with workload $W_{30}$, the numbers of claiming tries, which are between 1 and 3 as shown in Figure 5.7, are still quite low. For this workload we do no show the distribution of the placement times as with workload $W_{30}$ because the system was saturated and therefore unstable, during these experiments.




a. Without replication b. With replication

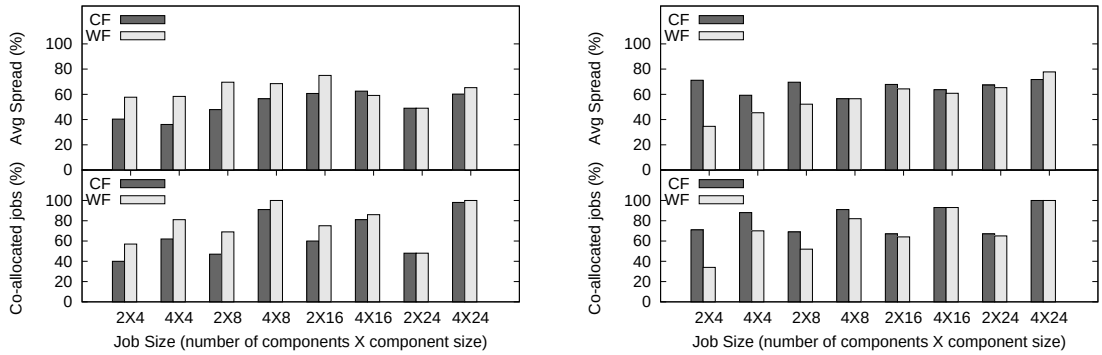
Figure 5.7: The average number of placement and claiming tries with workload $W_{50}$.

### 5.1.7 Assessing the level of co-allocation used

Our placement policies use co-allocation to achieve their primary goals: to minimize the file transfer times (CF), and to balance the numbers of idle processors (WF). Recall that both CF and WF are allowed to place different components of the same job on the same cluster. In order to assess the level of co-allocation actually used by a policy, we introduce a metric called the *job spread*, which for a job is defined as the ratio of the number of its execution sites and the number of its components (we will express it as a percentage). So if for a job of four components, all of its components are placed on different clusters, its job spread is 100%. On the other hand, if all of its components are placed on the same cluster, then its job spread is 25%.

We have done a separate set of experiments to study the average job spread and also the percentage of jobs that actually use co-allocation. For these experiments, we have created a new workload that utilizes 30% of the system and that consists of jobs with 2 or 4 components of equal sizes (number of processors), which can be 4, 8, 16 or 24. In this workload, each job component requires the same single file of size 2 GByte. Again,

we do two experiments, one without file replication and the other with files replicated in three different sites. The results of these experiments are shown in Figure 5.8.




a. Without replication b. With replication

Figure 5.8: The average job spread (upper graphs) and the percentages of jobs that use co-allocation (lower graphs).

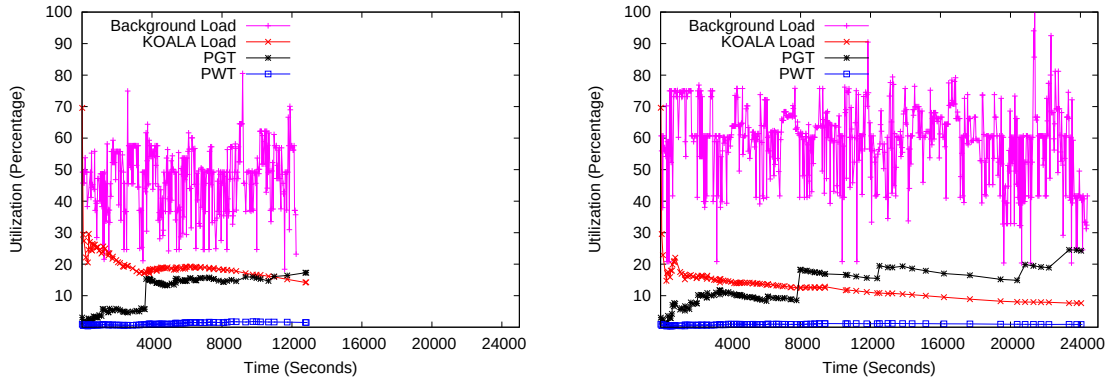
We first observe that with replication, CF places at least half of the components of the jobs on separate clusters, which is expected with CF because the number of potential execution sites increases with replication. Without replication, the average job spread of CF is slightly less compared to CF with replication because of the decrease in the number of potential execution sites. Second, the average job spread of WF is not affected by file replication, which can be explained by the fact that finding execution sites with WF depends on the numbers of available processors, and the size and the distribution of the background load across the sites. As a result of these two observations and with our background load, CF with replication uses co-allocation more compared to WF, while without replication, it is the opposite. Lastly, for the same total job size, the percentage of jobs using co-allocation increases as the number of job components increases.

### 5.1.8 The Close-to-Files policy with high background loads

It may be expected that the success of our workaround method for processor reservation by postponing the claiming of processors depends on the size and variation of the background load. In our previous experiments we observed that the average number of claiming tries, which was between 1 and 3, is quite low, indicating the success of our workaround method for processor reservation with a background load between 30% and 40%. However, this background load is fairly low. Therefore, to further test this method we performed experiments with workload $W_{30}$ with only CF and with replication while trying to maintain a background load of 50% or 60% (again employing dummy jobs as described in Section 5.1.3).

Figure 5.9 shows the results of these experiments. For both background loads, the KOALA load (co-allocation load) is much lower than 30%, and the experiments take (much) more time than expected (the last job arrives at about time 6500), and so the system is saturated. The real total utilization ranges between 75% and 80%, which is roughly equal to the utilization achieved in the experiments with $W_{50}$ in Section 5.1.6, when the system was also saturated.

A high background load is very bad for large jobs. In Figure 5.10.a, with 60% background load, the average placement time of jobs with 4 components of size 16 is about 3.3 hours! It should be noted that despite long placement times, all jobs finished successfully. Figure 5.10.b shows that the numbers of claiming tries are still quite low even with these much higher background loads, indicating the success of our workaround method for reservation. It should be noted that the increase in the number of claiming tries has the positive effect of reducing the PWT in favor of the PGT; the latter is now in the range of 7 to 20%.

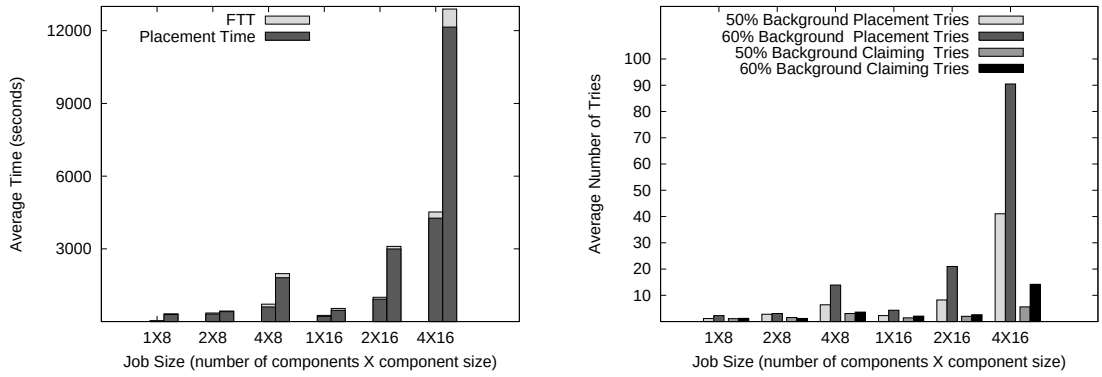



a. 50% background load b. 60% background load

Figure 5.9: The utilizations for the Close-to-Files placement policy with workload $W_{30}$ with different background loads.

## 5.2 An Evaluation of KOALA in an Unreliable Testbed

In this section we describe the experiments we have conducted to assess our co-allocation service in an unreliable environment. The experiments were done on the DAS-2 system immediately after a major upgrade of the operating system and the local resource manager (openPBS). Even though the system is homogeneous and centrally managed, it was then very unstable, and hence unreliable during the experiments. This gave us the opportunity to evaluate co-allocation with KOALA in a grid-like environment where the job failure rate is high. The fact that this rate is high even in such an environment shows the strong need for good fault tolerance mechanisms. Sections 5.2.1 and 5.2.2 describe the KOALA




a. 50% background load (left bars) and 60% background load (right bars)

b. The placement and claiming tries

Figure 5.10: The average Placement Time and File Transfer Time (left), and the number of placement and claiming tries (right) of the Close-to-Files placement policy with workload $W_{30}$ with different background loads.

setup and the workload used in this section. Sections 5.2.3–5.2.5 discuss the results of the experiments.

### 5.2.1 KOALA setup

In these experiments, again we did not impose limits on the number of placement and claiming tries. Like in the experiments of Section 5.1, the parameter $L$ is set at 0.75 but the interval between successive scans of the placement queues is increased to 4 minutes in order to decrease the excessive number of scans due to the unreliability of the testbed. The parameter $T$ of our claiming algorithm is set to 0, so we claim processors for any number components we can. Also, the decision to assign 0 to parameter $T$ was based on the variability of the number of processors available. Only jobs of high priority and low priority are used in these experiments, and in order to see the impact of job priorities we do not allow jobs to change their priorities by setting parameter $P$ of KOALA to $\infty$. The parameters $N_h$ and $N_l$ are set to 1, the parameters $n_1$ and $n_2$ are set to 1 and 2, respectively, and the parameters $n_3$ and $n_4$ are set to 1. This setup means we scan the high placement queue twice before we scan the low placement queue once. All of these KOALA parameters were explained in Section 4.2.1. The KRunner was used to submit jobs in these experiments as it was the only runner available at the time of these experiments. Also, we wanted to see how the KOALA scheduler copes with the unreliability of the DAS-2 testbed.

### 5.2.2 The workload

In these experiments, we put a workload of 500 jobs to be co-allocated on the DAS that all run the Poisson application described in Section 2.3.2, in addition to the regular workload of the ordinary users. In our experiments, we consider job sizes of 36 and 72, and four numbers of components, which are 3, 4, 6, and 8. The components request the same number of processors, which is obtained by dividing the job size by the number of components. We restrict the component sizes to be greater than 8, so the jobs of size 72 can have any of the four number of component, while those of size 36 have 3 or 4 components. All possible combinations of number of components and component sizes have the same probability. Each job component requires the same single file of either 4 or 8 GByte. The input files, which are randomly distributed, are replicated in two sites. The average execution time of this application with components of size 9, 12, 18, and 24 is 99.0, 127.0, 51.0, and 37.0 seconds, respectively, again independent of total job size. We assume the arrival process of our jobs at the submission site to be Poisson. At the start of the experiments, in the DAS clusters the number of processors were as shown in Table 5.2; however, the cluster of Leiden University was not available due to errors. Exactly repeating these experiments will be very difficult as the same experimental conditions in terms of the dynamic behavior of the DAS cannot be easily recreated.

Table 5.2: The numbers of processors in the DAS-2 clusters at the start of the experiments reported in Section 5.2.

| Cluster Location | Number of Processors |
|---|---|
| Vrije University | 138 |
| Leiden University | 44 |
| University of Amsterdam | 48 |
| Delft University | 62 |
| Utrecht University | 64 |

### 5.2.3 Utilization

At the start of the experiment, a total of 312 processors in 4 out of 5 clusters were available to KOALA for placing jobs. During the experiment, the cluster of Delft University reported a very high consecutive number of soft errors and was taken out of selection by KOALA. Soft errors are defined in Section 3.2.2 as execution-site specific errors caused by hardware or software faults of the execution sites. As a result, the number of processors available for selection was reduced to 250. The utilizations of these processors by jobs due to other DAS users and to KOALA are shown in Figure 5.11. In this figure we

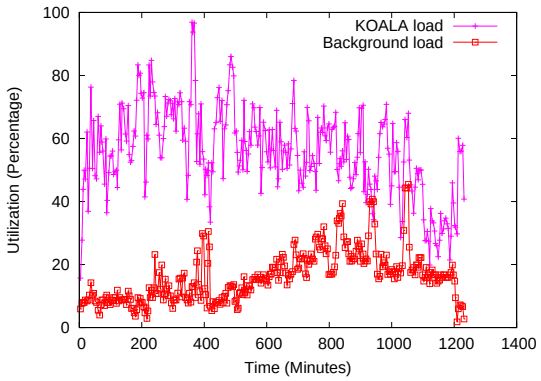


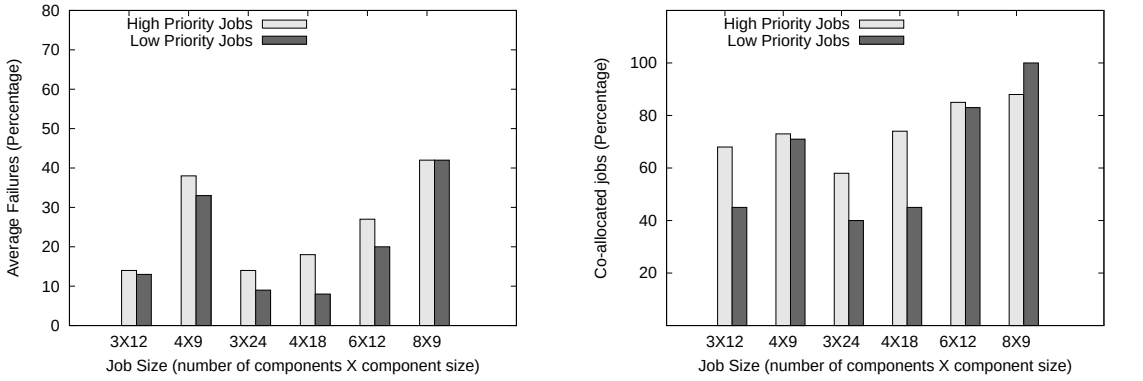

Figure 5.11: The system utilization during the experiment on an unreliable testbed.

see that 80% to 90% of the system was used during the experiment, which shows that co-allocation can drive the utilization to quite high levels.

### 5.2.4 Failures

The failures that we report in this section were caused by soft errors of the clusters. The errors in these experiments were due to bugs in the LRMS (OpenPBS) and to the incorrect configuration of some of the nodes. Since we are using a co-allocated workload, the failure of any of the components of a job causes the whole job to fail, and as a result, to be returned to the placement queue. Figure 5.12.a shows the average failure rate (expressed as a percentage) for each of the job sizes during the experiments. The failure rate of a job is calculated as the ratio of its number of failures due to soft errors and the number of its successful placement tries. By a successful placement try we mean a placement try that resulted in the job having started its execution. Since in the end, all jobs ran successfully, the number of failures of a job is equal to the number of successful placement tries minus one. The percentage of failures is much higher compared to the stable system, where it was always below 15%. From the figure, we observe more failures for high-priority jobs. This is expected because more attempts are performed to place, to claim, and therefore to run these jobs. As a result, more jobs are started simultaneously, which results in some components to be given mis-configured nodes because most of the time, these nodes are idle.

The percentage of jobs that were actually co-allocated, i.e., of jobs whose components were placed on multiple clusters, increases as the number of job components increases, as shown in Figure 5.12.b. It should be noted that the range of the percentage of jobs using co-allocation in this figure is the same as in Figure 5.8. Co-allocation has the effect of increasing the failure rate because then the chance for components to be placed on multiple clusters, and hence the chance of failures, increases.

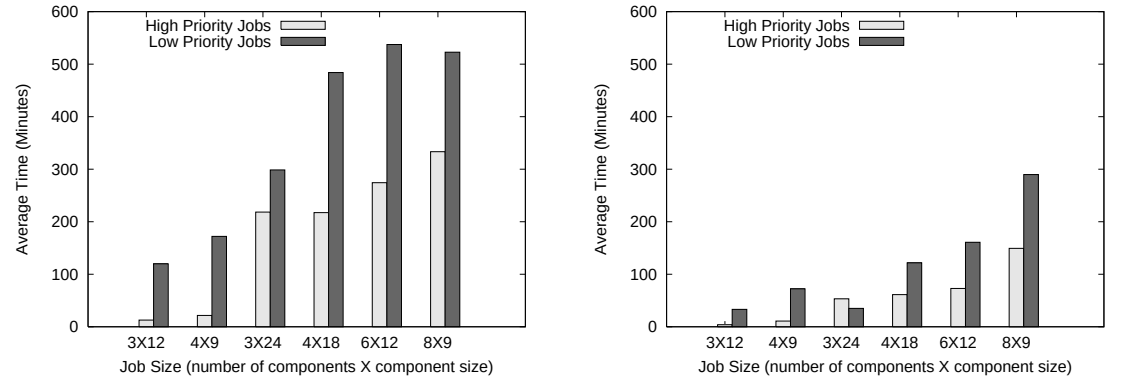



a. Percentages of failures of jobs b. Percentages of job that use co-allocation

Figure 5.12: The percentages of failures of jobs of different sizes, and the percentages of jobs that use co-allocation.

### 5.2.5 Placement Times and Start Delays

Failed jobs are returned to their respective placement queues in KOALA, which then tries to re-place these jobs until their execution succeeds. These re-placements result in multiple placement times, which we sum to get the total placement time of a job. The placement time and the total placement time were defined in Sections 4.2.1 and 4.2.2, respectively. In Figure 5.13.a we observe an increase in the total placement times of jobs as the number of components increases. The explanation for this is that with each re-placement, with the increase in the number of components, more time is likely to be spent waiting for clusters to have enough processors available simultaneously. Despite the increase in the total placement times, all our jobs eventually ran to completion successfully.




a. Total Placement Times b. Start Delays

Figure 5.13: The Total Placement Times and Start Delays of jobs.

Jobs of small sizes (e.g., of size 36 shown in Figure 5.13.a) do not suffer from long waiting times for enough processors to be available. Yet these jobs still require co-

allocation as shown in Figure 5.12.b, which helps to lower their total placement times. This is because components of jobs of smaller sizes are used by the scheduler to fill the "holes" left by the components with big sizes of other jobs.

Figure 5.13.b shows the start delays of jobs of different sizes, which are also affected by the number of failures and re-placements. As in the above observations, the start delay increases with the number of components, with high-priority jobs performing better than low-priority jobs.

Overall, we conclude that splitting jobs into more components does not necessarily lead to smaller total placement times (e.g., compare job sizes 3X12 and 4X9 in Figure 5.13.a). On the other hand, small jobs still require co-allocation to guarantee smaller total placement times and start delays. Nevertheless, we cannot conclude that jobs of smaller sizes perform much better, but rather we can conclude that co-allocation cannot avoid delaying considerably jobs requesting many processors in an unreliable testbed. Finally, also in an unreliable system, jobs of high priority out-perform jobs of low priority.

## 5.3 Relation with Simulation Studies of Co-allocation

Extensive simulation studies of processor co-allocation in multicluster systems such as the DAS have been performed [34, 36–40], and in this thesis, we present performance results of co-allocation obtained with KOALA in the DAS-2. This raises the question to what extent the results of these simulations and of our experiments can be compared, and if they can be compared, to what extent their results match or diverge. Below, we will first review the model used in those simulations, and then we will answer this question.

In the simulations, the influence of many parameters and properties of a model for processor co-allocation in multicluster systems has been investigated. These parameters and properties can be divided into three groups:

1. *Workload parameters*, which are:

   (a) The possible structures of the job requests considered are ordered (called fixed in this thesis), unordered (jobs consisting of multiple components that can go to any set of *different* clusters), or flexible;

   (b) The number and the sizes of the job components;

   (c) The runtimes of the jobs;

   (d) The communication overhead due to the wide-area communication when jobs are co-allocated;

   (e) The arrival process of the jobs, which is always assume to be Poisson.

The actual values of the parameters in (b)-(d) are either synthetic, based on traces of the DAS-2 or of the CTC workload from the Parallel Workload Archive [13], or derived from measurements of application runtimes on the DAS-2.

2. *System parameters*, which are:

   (a) The number and sizes of the clusters;

   (b) The heterogeneity of the system: All processors in the whole system are supposed to be identical.

3. *The properties of the queuing structure and the scheduling policies*, which are defined as follows:

   (a) The queueing structure in the system, which can either be only a single global queue, only local queues in the clusters through which both local single-component jobs and jobs that require co-allocation can be submitted, or a combination of both;

   (b) The priority structure when both local and global queues are present, with either the local queues or the global queue having priority over the other(s);

   (c) The queuing discipline dictating which job from a queue is eligible for scheduling, which is either First-Come-First-Served (FCFS) or backfilling (no service time estimates are used but a job can only be overtaken a limited number of times). However, if both global and local queues are present, the queuing discipline in both is FCFS;

   (d) The way the (local and/or global) schedulers are activated, which is event-based in that this is done when a job arrives or departs;

   (e) The scheduling policy for unordered co-allocated jobs, which can, among others, be Worst Fit.

The performance metrics used in the simulation studies are the average job response time (for the local and global queues separately if both are present) and the maximal utilization that can be achieved. However, the response time is only reported for all job sizes (in terms of the number of components and component size) together, and not for the different sizes separately.

We will now discuss the similarities and the differences between the model of multicluster systems in the simulations and the properties of KOALA and the DAS-2 while running the experiments. Our discussion will follow the lines of the division of the parameters into three groups as above for the simulations:

1. *Workloads*

(a) In the experiments with KOALA, only non-fixed job requests are used, the components of which are allowed to go to the same cluster, as opposed to unordered jobs in the simulations;

(b) The sizes of the jobs in the simulations are different from those in the experiments;

(c) The runtimes of the applications in the simulations are different from those in the experiments;

(d) The communication overhead due to the wide-area communication is of course included in the experiments as real jobs are executed in the DAS-2;

(e) In both the simulations and the experiments, the arrival process of jobs is Poisson.

2. *System*

(a) Except for a few simulations in which the influence of having different cluster sizes in the system is assessed, all simulation results are for a system with 4 clusters of size 32 each;

(b) All processors in the DAS-2 are identical.

3. *Queues and policies*

(a) In the experiments, there is a background load due to jobs submitted locally in the clusters through the local cluster managers in addition to the co-allocation workload submitted through KOALA, which corresponds to the queuing structure in the simulations with both a global queue and with local queues;

(b) With KOALA, jobs that need co-allocation are only placed on the system when the processors they require are immediately available, which means that in the terms of the simulation model, in the experiments the local queues have priority;

(c) In KOALA, the placement queues are scanned for any jobs that may fit on the system, which resembles the type of backfilling employed in the simulations. However, in KOALA there is no limit to the number of times a job in the (global) placement queues can be overtaken;

(d) Scheduling in KOALA is time-based in that the placement queues are scanned periodically;

(e) In the simulations, no input files or data co-allocation are considered, which means that WF is the only policy that has been considered in both settings.

From this comparison of the simulation model and the properties of KOALA and the DAS-2, we draw two conclusions. First, even though the simulation studies targeted the (idealized) operation of a system like the DAS with a co-allocation-enabled scheduler, the process of designing and implementing an actual scheduler in an actual system has led to design choices that make the actual scheduler deviate in many respects from the simulated scheduler. Secondly, we conclude that none of the instantiations of the simulation model corresponds well to the operation of KOALA in the DAS. On the one hand, we would have to take the simulation model with both global and local queues as there is background load in the DAS, but on the other hand, in that case FCFS is employed in the global queue in the simulations versus the version of backfilling in KOALA. Even if we argue that because we keep the background load in the DAS stable and so we can assume that the experimental results hold for a smaller system in which the parts of the clusters used by the local jobs are taken away, we still cannot make meaningful comparisons as the notions of unordered jobs in the simulations and of non-fixed jobs in the experiments are not the same.

## 5.4 Conclusions

In this chapter, we have evaluated the performance of the KOALA job policies that implement our co-allocation service. We have also presented the results of a performance and reliability test of KOALA while the DAS-2 testbed was unstable. The main conclusions of this chapter are as follows:

1. The KOALA scheduler operates correctly and reliably both in a stable and an unstable testbed.

2. The combination of the Close-to-Files placement policy and replication is beneficial.

3. In the absence of advance processor reservation in the LRMSs, the Incremental Claiming Policy can be used without wasting much processor time.

4. With co-allocation and with the job sizes similar to the ones used in this chapter, the utilization in a multicluster system like the DAS can be driven to about 80%.

5. Many jobs, including relatively small ones, use co-allocation when given the chance.

6. Even with high failure rates, KOALA succeeds in getting all jobs submitted to complete successfully.

# Chapter 6

# Evaluation of the KOALA Runners

In Section 3.3 we have presented the three KOALA runners which are currently operational, namely the KRunner, which is KOALA's default runner capable of running application types that do not have any special requirements, the DRunner, which is the specialized runner for grid-enabled Message Passing Interface (MPI) jobs, and the IRunner, which is designed to run Ibis applications. All of these runners support processor co-allocation, i.e., the spreading of applications across multiple sites in a grid. We have done experiments to evaluate the performance of these runners in the DAS-2 testbed and we present the results of these experiments in this chapter. We begin this chapter by describing the experimental setup in Section 6.1 followed by the results of these experiments in Section 6.2.

## 6.1 Experimental Setup

In this section we describe the setup of the experiments we have done to assess the operation of the runners. Section 6.1.1 presents two workloads that we impose on the DAS, and Section 6.1.2 describes the performance metrics that we record during the experiments.

### 6.1.1 The workloads

In our experiments, we use two workloads, $W_{nc}$ and $W_c$, which are a non co-allocated and a co-allocated workload, respectively. The jobs in these workloads have job sizes and numbers of components as shown in Table 6.1. Applications can have any of the job sizes in this table, and in $W_c$ any of the corresponding number of components. The components are of equal size, which is obtained by dividing the job size by the number of components. For a single job, its size (number of processors) and number of components are picked at random and uniformly. Based on this, we generate the two workloads $W_{nc}$ with 400 non co-allocated jobs and $W_c$ with 400 co-allocated jobs.

Table 6.1: The job sizes and the corresponding numbers of components in the two workloads $W_{nc}$ and $W_c$.

| Job Size | Number of Components | |
|---|---|---|
| | $W_{nc}$ | $W_c$ |
| 8 | 1 | 2, 4 |
| 32 | 1 | 4, 8 |
| 48 | 1 | 6, 12 |

In this experiments there is the workload of the ordinary users (background load), which again we try to maintain between 30% and 40% as described in Section 5.1.3. We have run the two workloads twice on the DAS. In the first run, each job in the workloads runs one of the three MPI applications, the Poisson application, the Fiber Tracking (FT), and the Lagrangian Particle Model (LPM) described in Section 2.3.2, and in the second run, each job runs one of the three Ibis applications, the N-Queens, the Raytracer, and the red/black Successive Over Relaxation (SOR) application, also described in Section 2.3.2. An application for each job in each run is picked at random and uniformly. The MPI jobs with a single component in $W_{nc}$ are non-grid enabled and therefore are submitted with the KRunner. The MPI jobs in $W_c$ are grid-enabled and therefore, they are submitted with the DRunner. All the Ibis jobs are submitted with the IRunner. As a consequence, we test all three runners in our experiments. We assume the arrival process of our jobs to be Poisson with a mean arrival rate of 2.4 jobs per minute.

The experimental conditions for workload generation, workload submission, and the background load stated above together with the application runtimes as reported in Section 6.2.1 are sufficient for repeating our experiments.

### 6.1.2 Performance metrics

In total we perform four experiments, one for each workload for either MPI or Ibis jobs. During the experiments we record the following performance metrics:

- The **runtime**, which is the duration of the execution from when a job is started to its termination.
- The **throughput**, i.e., the number of successfully started jobs per unit of time.
- The **cumulative number of jobs**, which is the accumulated number of jobs that have run and the ones which are still running up to a certain time within the makespan of the experiment.
- The DAS **utilization** due to background jobs and due to the jobs in our experiment separately.

- The application **Start Time Overhead** (STO), which is the overhead incurred from when the runner starts deploying a job for execution until the time the job is actually running. This overhead also includes the time that the job spends being processed by the LRMS (SGE in our case). It should be noted that our jobs are never queued in the LRMSs because KOALA places the job components on execution sites with enough idle processors to execute the jobs immediately.

- The **number of failures**, which is the number of failures due to soft errors resulting in a job being returned to the KOALA scheduler and its placement being retried.

## 6.2 Performance Results

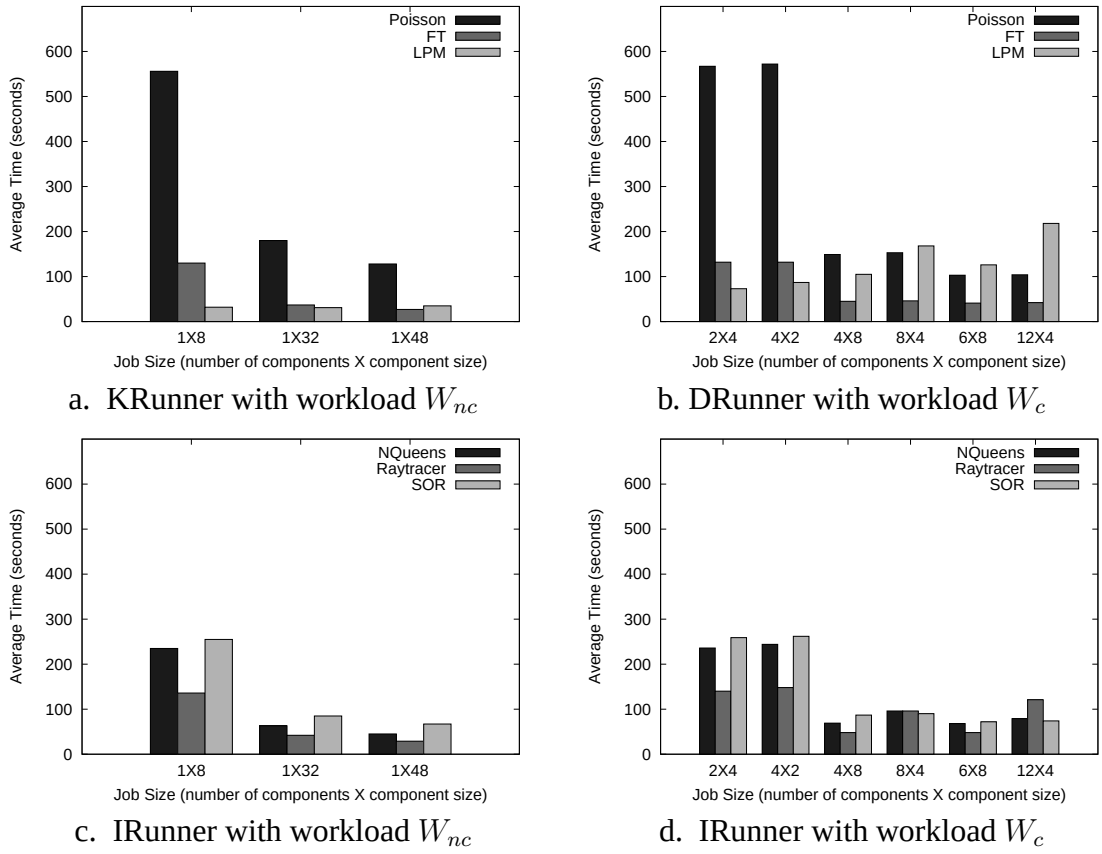
In Sections 6.2.1–6.2.5 we present the results of the experiments we have conducted on the DAS to assess the performance of the KOALA runners. During the experiments, the DAS clusters had the numbers of processors as shown in Table 6.2. During our experiments there were jobs of other DAS users using some of these processors, which resulted in only two clusters being able to run jobs of size 48.

Table 6.2: The numbers of the processors in the DAS-2 clusters at the start of the experiments reported in Section 6.2.

| Cluster Location | Number of Processors |
|---|---|
| Vrije University | 134 |
| Leiden University | 36 |
| University of Amsterdam | 32 |
| Delft University | 60 |
| Utrecht University | 50 |

### 6.2.1 Runtimes

Figure 6.1 shows the average runtimes of the applications during the experiments with workloads $W_{nc}$ and $W_c$. As expected in parallel processing, the average runtimes of the applications decrease as the number of processors goes up. For the LPM application submitted in workload $W_c$, its runtime increases considerably with the increase in the number of components. For instance, running this application with 4 components of size 12 instead of with a single component of size 48, makes the average runtime go from 35 to 218 seconds. This is because the LPM application is communication intensive (with many "many-to-many" communication patterns) and therefore, its average runtime is affected




a. KRunner with workload $W_{nc}$

b. DRunner with workload $W_c$

c. IRunner with workload $W_{nc}$

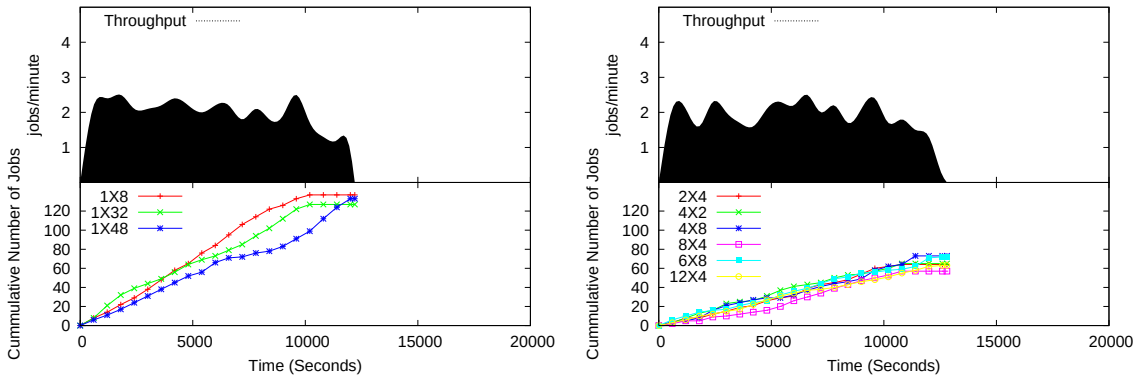
d. IRunner with workload $W_c$

Figure 6.1: The average runtimes of the applications in workloads $W_{nc}$ and $W_c$ submitted with the runners.

by the slow wide-area communication. Also the average runtime of the Satin jobs (N-Queens and Raytracer) with many components (8 and 12) is affected by the slow wide-area communication. This is because in Satin, the work is distributed across the processors by work stealing: when a processor runs out of work, it picks another processor, possibly in a different cluster, at random and steals a job from it. Of the two Satin applications, the Raytracer application is affected the most by the slow wide-area communication since it sends more data [96].

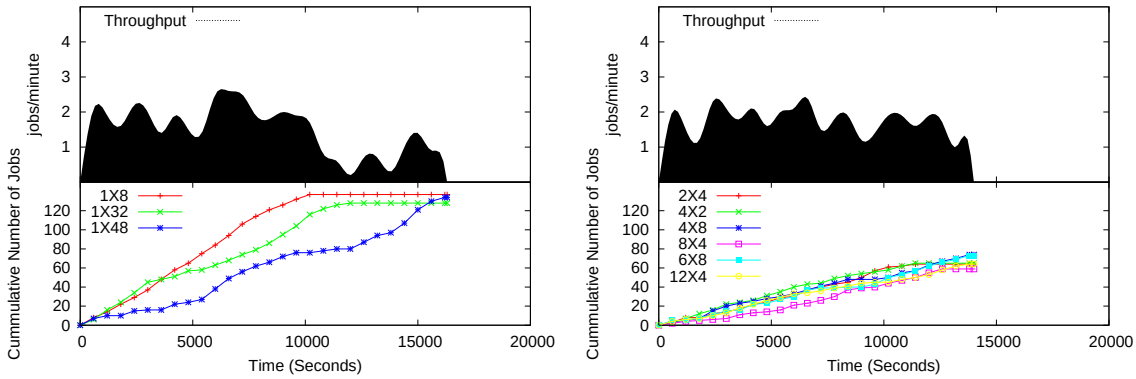
### 6.2.2 Throughput and cumulative number of jobs

Figure 6.2 shows the throughputs and the cumulative numbers of jobs of the three runners with workloads $W_{nc}$ and $W_c$. In these experiments, the last job was submitted at 10800 seconds, which is somewhat higher than the expected time of 10000 seconds. This is due to the CPU load of the submission site which, when it reaches a certain threshold, deliberately delays the job submissions to decrease this load. From the figure we observe




a. KRunner with workload $W_{nc}$

b. DRunner with workload $W_c$




c. IRunner with workload $W_{nc}$

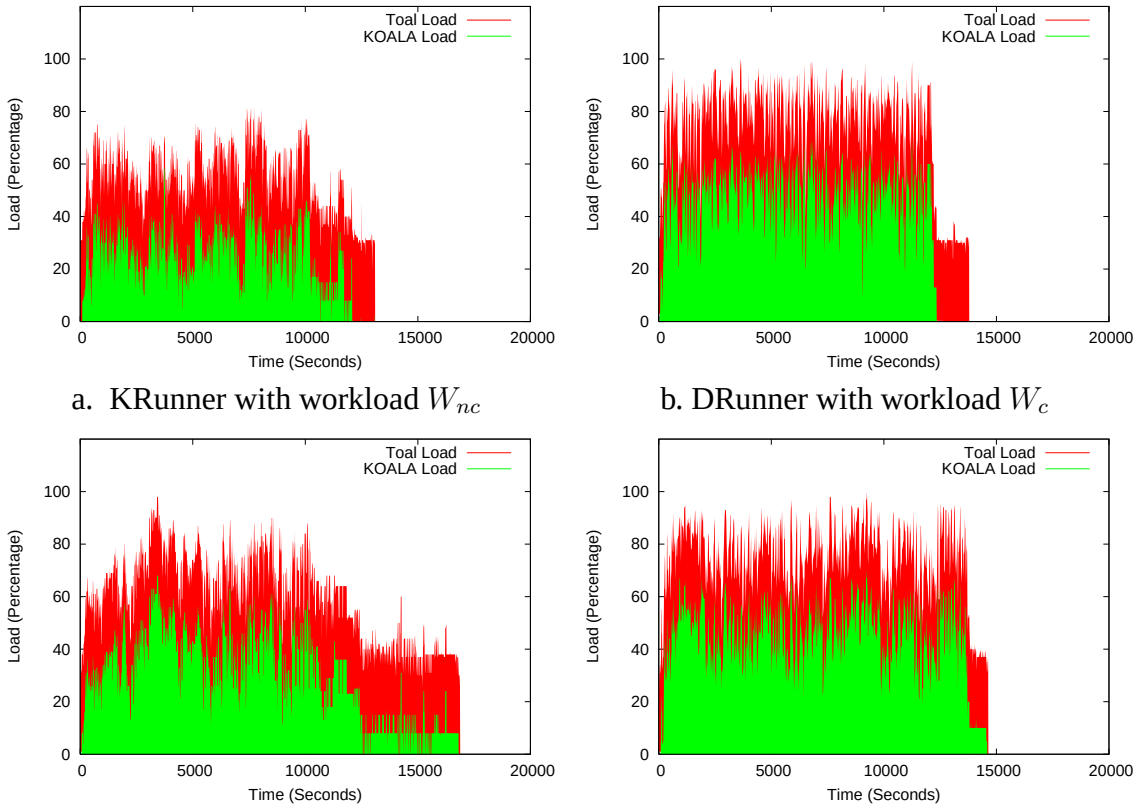
d. IRunner with workload $W_c$

Figure 6.2: The throughput (upper graphs) and the cumulative number of jobs (lower graphs) of the runners with workloads $W_{nc}$ and $W_c$.

that the average throughput of both the KRunner and the DRunner is around 2 jobs per minute, which means that the system is close to being stable.

For the IRunner with workload $W_{nc}$ after the submission of the last job, the throughput falls below a job per minute. The reason for the low throughput is the way the background load is distributed among the clusters. During this experiment, the local jobs that we did not have control over were distributed in such a way that only one job of size 48 could run at a time. During the experiments with this workload, application-specific errors were observed which caused further delay in executing jobs of size 48. Figure 6.2.c clearly shows that at the end of the job submissions, only jobs of size 48 are still in the queue.

The IRunner with workload $W_c$ has a throughput of slightly less than 2 jobs per minute. This is caused by the components of the Satin jobs that have failed to join their respective computations and end up running as redundant copies of jobs. As a result, processors are held longer than expected and therefore, the throughput decreases. Also, during this run, application-specific errors were observed which caused applications to hang while holding processors. The jobs of these applications were eventually aborted by




a. KRunner with workload $W_{nc}$

b. DRunner with workload $W_c$

c. IRunner with workload $W_{nc}$

d. IRunner with workload $W_c$

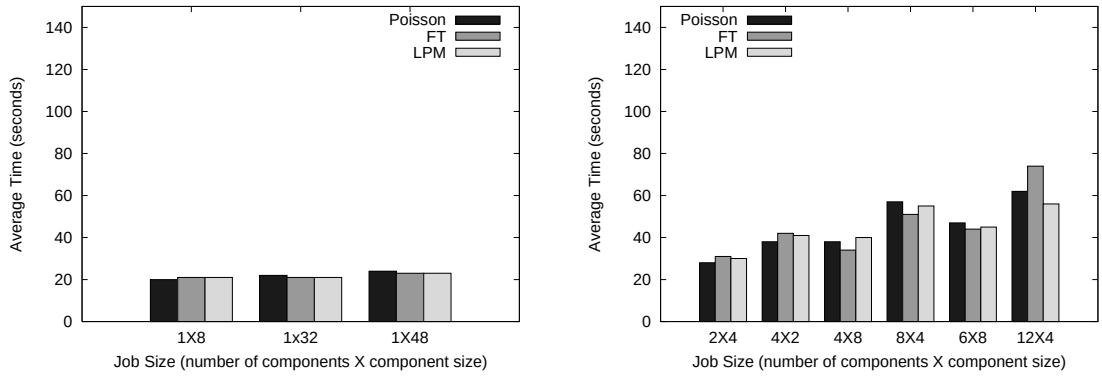
Figure 6.3: The utilizations of the system for the different runners with workloads $W_{nc}$ and $W_c$.

KOALA after a runtime of 15 minutes, which is the default for all jobs submitted to the DAS, has expired.

The cumulative numbers of jobs for workload $W_c$ show that the jobs of all sizes are executed equally, which shows the advantage of co-allocation. The situation is different for workload $W_{nc}$ because of the same reason given above for the throughput.

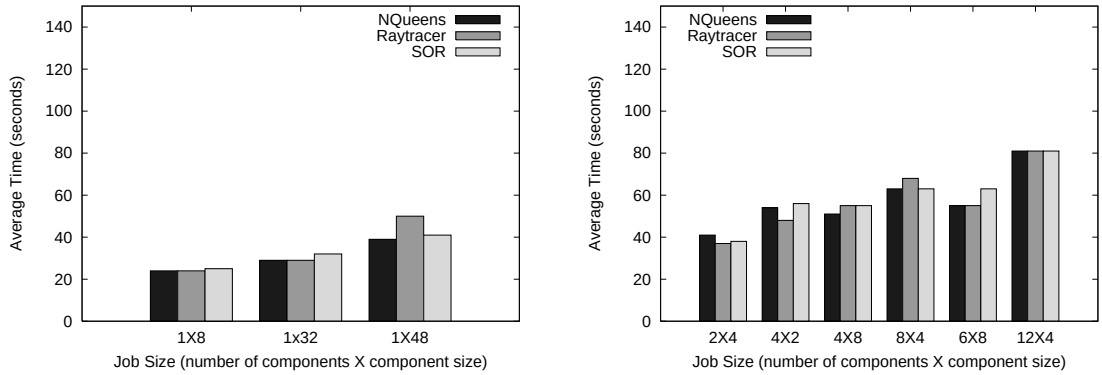
### 6.2.3 Utilization

Figure 6.3 shows the utilizations of the DAS for the three runners with workloads $W_{nc}$ and $W_c$. In this figure we see that the total load with workload $W_c$ varies between 70% and 80%, which is higher than the total load with workload $W_{nc}$, which varies between 60% and 70%. This is because components of the co-allocated workload $W_c$ have smaller sizes and therefore, are more easily placed by the KOALA scheduler. The components with smaller sizes are used by the scheduler to fill the “holes” left by the components with big sizes. As a result, the utilizations with workload $W_c$ are high compared to those with $W_{nc}$.




a. KRunner with workload $W_{nc}$

b. DRunner with workload $W_c$




c. IRunner with workload $W_{nc}$

d. IRunner with workload $W_c$

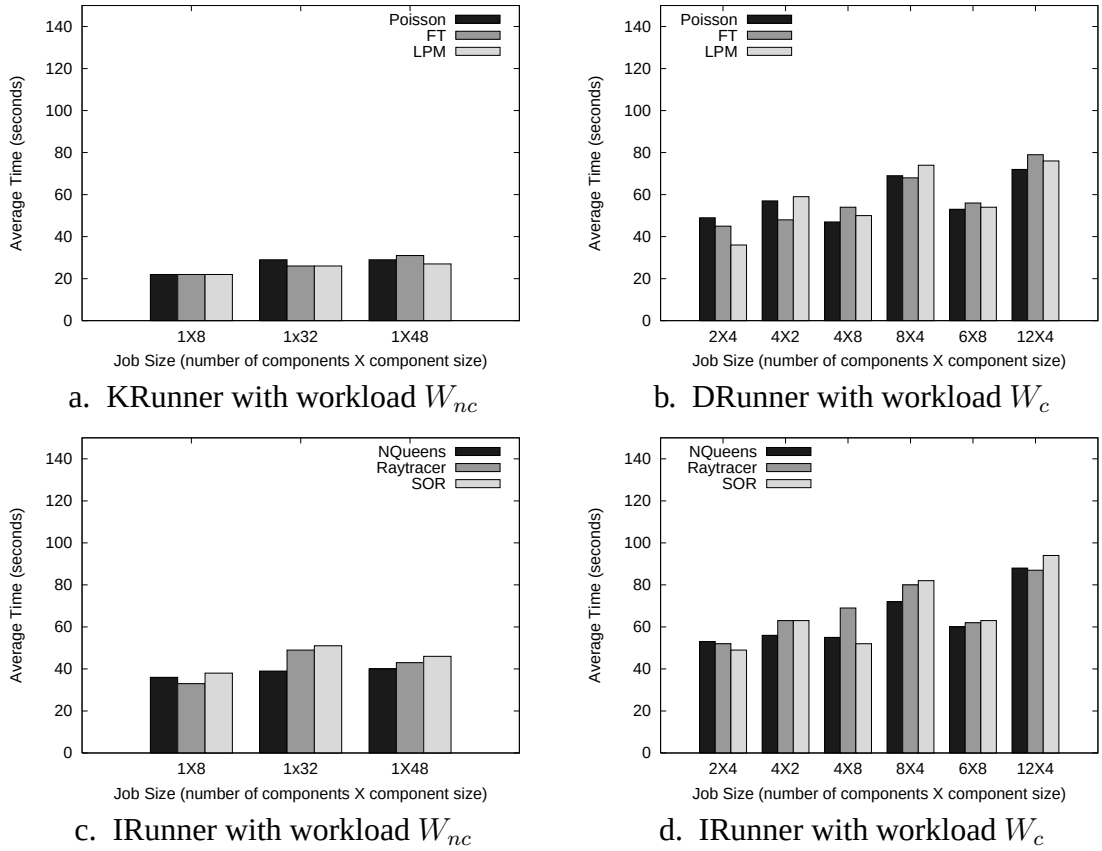
Figure 6.4: The average Start Time Overhead of the runners with workloads $W_{nc}$ and $W_c$.

For the IRunner with both workloads $W_{nc}$ and $W_c$, the experiments run longer because of the lower throughputs. Also, the longer tail in Figure 6.3.c for the IRunner with workload $W_{nc}$ is evidence of the application-specific errors explained in Section 3.2.2.

### 6.2.4 Start Time Overhead

The Start time Overhead (STO) is an important metric for the runners since it reports the delay of the execution of jobs caused by the middleware. In Figure 6.4 we observe that the average STO increases with the number of components per job size with workload $W_c$. The increase in the number of components causes an increase in the number of GRAM instances at the head node of the submission site as well as an increase in the number of GRAM job managers at the head nodes of the execution sites. This means that the head nodes are being used heavily and therefore, their response times are correspondingly slow. As a result, there are delays in the synchronization of the start of job components, which results in an increase in the STO. The IRunner with workload $W_c$ has an average STO that is slightly higher than for the DRunner since the nameservers (one for each running job)

are also running at the head node of the submission site. For workload $W_{nc}$, the average STO is much smaller.




a. KRunner with workload $W_{nc}$

b. DRunner with workload $W_c$

c. IRunner with workload $W_{nc}$

d. IRunner with workload $W_c$

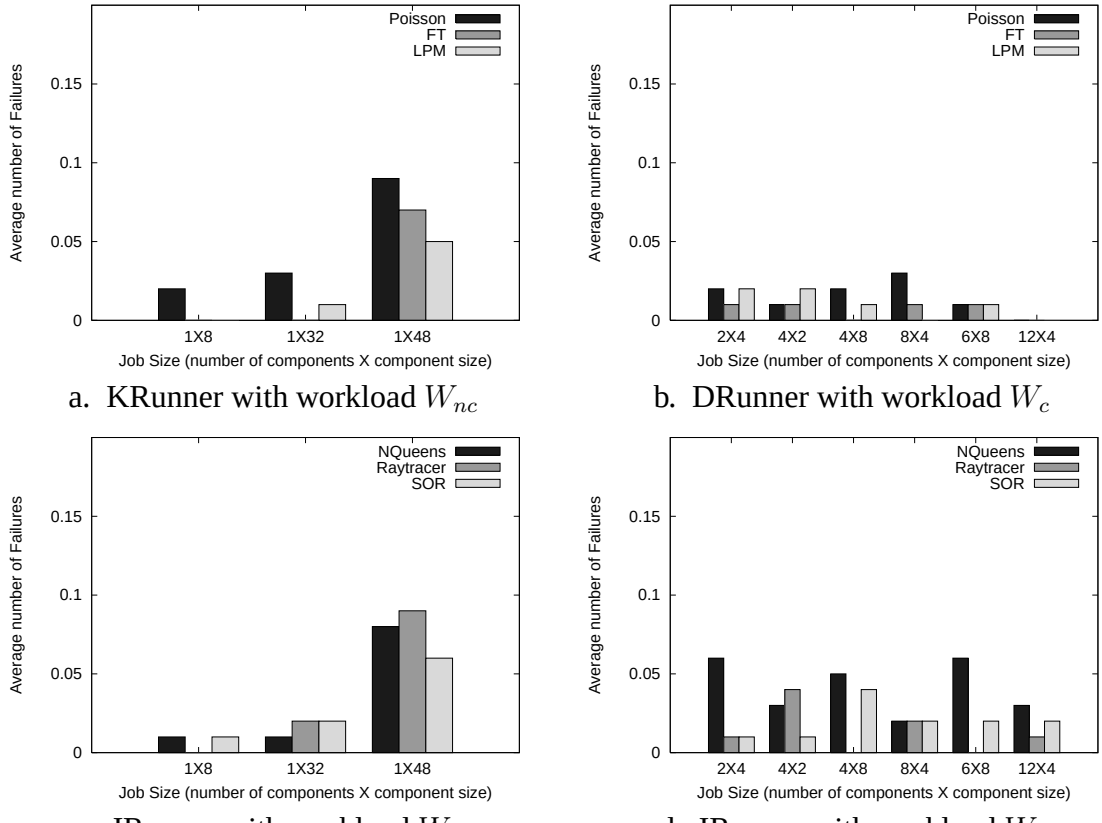
Figure 6.5: The average Start Time Overhead of the runners with workloads $W_{nc}$ and $W_c$ with a mean arrival rate of 4 jobs per minute.

The above experiments were done in a stable system. In order to make observations of the average STO in an unstable system, we have repeated the experiments with an arrival rate of 4 jobs per minute instead of 2.4. At this arrival rate we observe no increase in the throughput of the runners compared to the throughput with an arrival rate of 2.4 jobs per minute, which indicates that the system is now unstable. Despite observing the same throughput as in a stable system, the average STO has increased for both workloads as can be seen in Figure 6.5. This is to be expected because there are more runner instances of jobs that are waiting to be placed, and of jobs that are executing at the head node of the submission site, which increases the load of the head node.

### 6.2.5 Number of failures

The average number of failures per job we observed during the experiments with the runners reported on above is less than 0.05, which is very low. To observe better the number of failures, we repeated the experiments multiple times over a period of one month (in total 16,000 jobs were submitted). We show the average of the number of failures per job for all job sizes in Figure 6.6. Again, we emphasize that all jobs successfully ran to completion in the end.

Overall, these numbers are very low. This is because the soft errors that caused failures were solved easily by the KOALA fault tolerance mechanisms (see Section 3.2.2). These mechanisms were not so successful with jobs of size 48 in workload $W_{nc}$ because failed jobs of this size were usually restarted on the same cluster that they previously ran on. Clearly, this problem did not occur for co-allocated jobs of total size 48, which again shows the benefit of co-allocation.




a. KRunner with workload $W_{nc}$

b. DRunner with workload $W_c$

c. IRunner with workload $W_{nc}$

d. IRunner with workload $W_c$

Figure 6.6: The average number of failures per job over the period of one month (the failure rates for all job sizes are shown, but some are (very close to) 0).

## 6.3 Conclusions

In this chapter, we have presented the experiments done to assess the KOALA runners on the DAS-2 testbed. The main conclusions of this chapter are as follows:

1. The runners operate correctly in the DAS testbed.
2. Jobs of all sizes are executed equally when co-allocation is used, whereas without co-allocation large jobs may be delayed considerably.
3. The overhead in the middleware when starting jobs is strongly influenced by the load of the head nodes of the clusters on which this middleware runs.
4. The KOALA fault tolerance mechanisms minimize the number of failures caused by hardware and software faults of the execution sites.

# Chapter 7

# Conclusion

In this thesis, we have studied the problem of co-allocation in grids, i.e., the allocation of both processors and data to single applications in multiple sites and the simultaneous access to these resources by the applications. As stated in Chapter 1, the co-allocation problem leads to the following challenges of grid resource management: allocating resources in multiple sites, guaranteeing the simultaneous availability of the co-allocated resources when they are about to be accessed by the applications, and managing sets of highly dynamic grid resources. Also, we have addressed the challenge of automating the deployment of different application types on the grid, which is difficult because of the characteristics of the grid applications and of the grid infrastructure.

In Section 7.1 we summarize the approach we have used to address the mentioned challenges. We present our conclusions in Section 7.2 and finally, in Section 7.3 we describe open research questions.

## 7.1 Approach

In order to deal with the challenges of grid resource management and of deploying grid applications, we have developed KOALA, a Grid Resource Management System, we have deployed it on the DAS, and we have done extensive experiments with it. KOALA has proven to be working reliably on the DAS testbed with over 500,000 jobs already submitted with it. The architecture of KOALA consists of two major layers, namely, the KOALA scheduler and the runners, which are job submission and monitoring tools. When designing KOALA, arriving at the two layers of the scheduler and the runners was not straightforward; the development of KOALA started with a monolithic single-layered scheduler called PDCA on the DAS-2. However, the need to increase our research domain to include research in grid application types such as workflows, Parameter Sweep Applications (PSAs) and Bags-of-Tasks (BoTs), and in grid benchmarking, forced us into re-designing and re-implementing KOALA using the layered approach, which has proved to be more

flexible, stable, and reliable than the monolithic structure we started with. With this approach, adding support for new application types has now been easy and does not disturb the operation of KOALA, contrary to our initial design.

The KOALA scheduler addresses the challenge of allocating resources in multiple sites with its Close-to-Files (CF) and Worst Fit (WF) policies. CF has been designed to minimize the long delays when starting a job because of long input file transfers by selecting the execution sites of its components close to sites where their input files are located. On the other hand, the WF policy has been designed to balance processor usage among the clusters and if possible, to minimize file transfer times as well. To guarantee the simultaneous availability of the co-allocated resources, the KOALA scheduler has the Incremental Claiming Policy (ICP), which is used in the absence of support for advance processor reservation by the Local Resource Management Systems. ICP tries to make processors available for job components, if necessary by finding processors at other sites than previously allocated or, if permitted, by forcing processor availability through the preemption of running jobs.

The runners framework addresses the challenge of deploying grid applications through the use of the KOALA runners, which are job submission and monitoring tools. Due to the modularity of the design of the runners framework, different runners can be written to support the unique characteristics of different applications with ease. The runners framework has been designed with fault tolerance mechanisms that deal with the reliability issues of the grid infrastructure. The runners framework and the scheduler work together to manage sets of highly dynamic grid resources.

KOALA, which has been operational on the DAS-2 testbed since September 2005 and on the DAS-3 since May 2007, has been used successfully in different projects on the DAS. Because of its modular structure and its ease of use, KOALA has become a tool to be used in new research. For example, at the time of writing of this thesis, there is ongoing work in the University of Amsterdam in extending KOALA to support the scheduling of light paths of the optical network in order provide more bandwidth when transferring large files. Other ongoing work with KOALA includes the development of a Grid Application Toolkit (GAT) [1] adaptor for KOALA at the Vrije University, and the addition of cycle scavenging support to KOALA.

## 7.2 Conclusions

Based on the research reported in this thesis, we draw the following major conclusions:

1. Co-allocation is a useful mechanism in Grid Resource Management Systems. With the use of co-allocation, it is possible to run jobs that require more processors than are available at any single grid site, or jobs that require resources distributed at

different sites. On the down side, wide-area communication still has a huge impact on the runtimes of communication-intensive applications. However, with the rapid advance in optical network technology, this problem may soon be manageable.

2. It is possible to implement reliable co-allocation mechanisms in a Grid Resource Management System.

3. When developing a Grid Resource Management System, a layered architecture is recommended over a monolithic single-layered scheduler. With a layered architecture, support for new application types can be added with ease and without disturbing the operation of the Grid Resource Management System.

4. Based on a layered architecture, running grid applications is simplified by separating application scheduling from application deployment. This allows application programmers to focus on application development without worrying about grid resource management.

5. Grid resources are hardly reliable, which is evident even in a homogeneous testbed like DAS-2, which is centrally managed. Therefore, designing reliable mechanisms and studying them through experimentation in real environments, is very important.

6. Our experiences obtained by running experiments have shown the correct and reliable operation of KOALA with its co-allocation and job deployment mechanisms, both in stable and unstable testbeds.

## 7.3 Open Research Questions

Although KOALA has been tested thoroughly and is currently a fully operational Grid Resource Management System, the following five issues still need to be addressed:

1. Our scheduler design has been restricted to a single global scheduler which may introduce a bottleneck like any centralized component deployed in larger grids than the DAS. Approaches that consider distributed global grid schedulers need to be investigated.

2. Our scheduling policies work well in a homogeneous environment. In a heterogeneous environment, these policies need to be extended to include more parameters such as processor speed when making scheduling decisions.

3. An extensive performance study of the policies and the KOALA scheduler is required in a heterogeneous environment such as when interconnecting the DAS-3 and Grid'5000 [43].

4. Different approaches for data scheduling are still required in KOALA, e.g., segmenting large input files and scheduling the transfers of only the chunks required by the job components as a means to minimize the transfer times and the storage space.

5. More runners for other application types that we are currently not supporting need to be added.

6. The fault tolerance mechanisms in the runners framework need to be extended to tolerate more types of faults, e.g., tolerating submission site crashes and file-transfer failures. In addition, a more extensive performance study of the runners, preferably in a heterogeneous grid environment, is required.

# Bibliography


[1] A Grid Application Toolkit and Testbed. http://www.gridlab.org/.

[2] BitTorrent.org. http://www.bittorrent.org/.

[3] Condor High Throughput Computing. http://www.cs.wisc.edu/condor/.

[4] DIET Grid Middleware. http://graal.ens-lyon.fr/DIET/.

[5] GrenchMark: Grid Benchmark Tool. http://grenchmark.st.ewi.tudelft.nl.

[6] Grid Service Broker: A Grid Scheduler for Computational and Data Grids. http://www.gridbus.org/broker/.

[7] Grid'5000 official web site. https://www.grid5000.fr.

[8] GridFTP tests on DAS-2 clusters. http://trafficlight.uva.netherlight.nl/GridFTP-tests/Intro.html.

[9] GROMACS: Fast, Free and Flexible MD. http://www.gromacs.org/.

[10] Iperf Version 1.7.0. http://dast.nlanr.net/Projects/Iperf/.

[11] KOALA Co-Allocating Grid Scheduler. http://www.st.ewi.tudelft.nl/koala.

[12] Lightweight Middleware for Grid Computing. http://glite.web.cern.ch/glite/.

[13] Logs of Real Parallel Workloads from Production Systems. http://www.cs.huji.ac.il/labs/parallel/workload/logs.html.

[14] Mathematical Modeling of Physical Phenomena. http://ta.twi.tudelft.nl/wagm/.

[15] Maui Scheduler. http://supercluster.org/maui/.

[16] Molecular Dynamics in Real-time Immersive Virtual Environment. `http://visualization.tudelft.nl/Projects/MolDrive`.

[17] Mpich-g2. `http://www3.niu.edu/mpi/`.

[18] Myri-10G Overview. `http://www.myri.com/Myri-10G/overview/`.

[19] Open Grid Forum. `http://ogf.org`.

[20] Philips Research. `http://www.research.philips.com/`.

[21] The Distributed ASCI Supercomputer (DAS). `http://www.cs.vu.nl/das2`.

[22] The Globus Toolkit. `http://www.globus.org/`.

[23] The Portable Batch System. `http://www.pbspro.com`.

[24] The Portable Batch System. `www.openpbs.org`.

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

[26] The Sun Grid Engine (SGE). `http://gridengine.sunsource.net/`.

[27] D. Abramson, R. Buyya, and J. Giddy. A Computational Economy for Grid Computing and its Implementation in the Nimrod-G Resource Broker. *Future Generation Computer Systems*, 18(8):1061–1074, 2002.

[28] W. Allcock, J. Bresnahan, I. Foster, L. Liming, J. Link, and P. Plaszczac. GridFTP Update. Technical report, Globus Alliance, 2002.

[29] Giovanni Aloisio, Massimo Cafaro, Italo Epicoco, Sandro Fiore, Daniele Lezzi, Maria Mirto, and Silvia Mocavero. iGrid, a Novel Grid Information Service. In *Proc. of the European Grid Conference*, volume 3470 of *Lecture Notes in Computer Science*, pages 506–515. Springer-Verlag, 2005.

[30] M. V. Ameijden. Molecular Dynamics Simulation and Visualisation in a Grid Environment. Master's thesis, Delft University of Technology, 2006.

[31] S. Ananad, S. Yoginath, G. von Laszewski, and B. Alunkal. Flow-based Multistage Co-allocation Service. In *Proc. of the International Conference on Communications in Computing*, pages 24–30, 2003.

[32] D. Azougagh, J. Yu, and S. R. Maeng. Resource Co-Allocation: A Complementary Technique that Enhances Performance in Grid Computing Environment. In *Proc. of the 11th International Conference on Parallel and Distributed Systems (ICPADS'05)*, pages 36–42, 2005.

[33] F. Azzedin, M. Maheswaran, and N. Arnason. A Synchronous Co-Allocation Mechanism for Grid Computing Systems. *Cluster Computing*, 7:39–49, 2004.

[34] S. Banen, A.I.D. Bucur, and D.H.J. Epema. A Measurement-Based Simulation Study of Processor Co-Allocation in Multicluster Systems. In *Proc. of the 9th Workshop on Job Scheduling Strategies for Parallel Processing*, volume 2862 of *LNCS*, pages 105–128. 2003.

[35] A.I.D. Bucur. *Performance Analysis of Processor Co-Allocation in Multicluster Systems*. PhD thesis, Delft University of Technology, 2004.

[36] A.I.D. Bucur and D.H.J. Epema. The Influence of Communication on the Performance of Co-Allocation. In *Proc. of the 7th Workshop on Job Scheduling Strategies for Parallel Processing*, volume 2221 of *LNCS*, pages 66–86. 2001.

[37] A.I.D. Bucur and D.H.J. Epema. An Evaluation of Processor Co-Allocation for Different System Configurations and Job Structures. In *Proc. of the 14th Symposium on Computer Architecture and High Performance Computing*, pages 195–203, 2002.

[38] A.I.D. Bucur and D.H.J. Epema. The Maximal Utilization of Processor Co-Allocation in Multicluster Systems. In *Proc. of International Parallel and Distributed Processing Symposium (IPDPS)*, pages 60–69, 2003.

[39] A.I.D. Bucur and D.H.J. Epema. The Performance of Processor Co-Allocation in Multicluster Systems. In *Proc. of the 3rd IEEE/ACM International Symposium on Cluster Computing and the GRID (CCGrid2003)*, pages 302–309, 2003.

[40] A.I.D. Bucur and D.H.J. Epema. Trace-Based Simulations of Processor Co-Allocation Policies in Multiclusters. In *Proc. of the 12th IEEE International Symposium on High Performance Distributed Computing (HPDC-12)*, pages 70–79, 2003.

[41] A.I.D. Bucur, R. Kootstra, and R. G. Belleman. A Grid Architecture for Medical Applications. In *Proc. of the third Healthgrid conference*, pages 127–137, 2005.

[42] J. Buisson, H.H. Mohamed, O. Sonmez, W. Lammers, and D.H.J. Epema. Scheduling Malleable Applications in Multicluster Systems. In *Proc. of the IEEE International Conference on Cluster Computing 2007*, pages 372–381, 2007.

[43] F. Cappello and H.E. Bal. Toward an International Computer Science Grid. In *Proc. of the 6th IEEE/ACM International Symposium on Cluster Computing and the GRID (CCGrid2007)*, pages 3–12, 2007.

[44] F. Cappello, E. Caron, M. Dayde, F. Desprez, E. Jeannot, Y. Jegou, S. Lanteri, J. Leduc, N. Melab, G. Mornet, R. Namyst, P. Primet, and O. Richard. Grid'5000: A Large Scale, Reconfigurable, Controlable and Monitorable Grid Platform. In *Proc. of the 6th IEEE/ACM International Workshop on Grid Computing Grid'2005*, pages 99–106, 2005.

[45] B. Caprita, W. C. Chan, J. Nieh, C. Stein, and H. Zheng. Group-Ratio Round Robin: O(1) Proportional Share Scheduling for Uniprocessor and Multiprocessor Systems. In *Proc. of the 2005 USENIX Annual Technical Conference*, pages 337–352, 2005.

[46] E. Caron and F. Desprez. DIET: A Scalable Toolbox to Build Network Enabled Servers on the Grid. *Journal of High Performance Computing Applications*, 20(3):335–352, 2006.

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

[48] H. Casanova, G. Obertelli, F. Berman, and R. Wolski. The AppLeS Parameter Sweep Template: User-Level Middleware for the Grid. In *Proc. of the IEEE/ACM conference on Supercomputing*, pages 75–76, 2000.

[49] Platform Computing. Open Source Metascheduling for Virtual Organizations with the Community Scheduler Framewok (CSF). Technical report, Platform Computing, 2003.

[50] K. Czajkowski, I. Foster, C. Kesselman, V. Sander, and S. Tuecke. SNAP: A Protocol for Negotiating Service Level Agreements and Coordinating Resource Management in Distributed Systems. In *Proc. of the 8th Workshop on Job Scheduling Strategies for Parallel Processing*, volume 2537 of *LNCS*, pages 153–183. 2002.

[51] Karl Czajkowski, Ian T. Foster, and Carl Kesselman. Resource Co-Allocation in Computational Grids. In *Proc. of the 8th IEEE International Symposium on High Performance Distributed Computing (HPDC-8)*, pages 219–228, 1999.

[52] H. Dail, F. Berman, and H. Casanova. A Decoupled Scheduling Approach for Grid Application Development Environments. *Journal of Parallel and Distributed Computing*, 63(5):505–524, 2003.

[53] M. Danelutto, M. Vanneschi, C. Zoccolo, and T. Tonellotto. HPC Application Execution on Grids. In *Proc. of the Workshop on Future Generation Grids*, CoreGRID series, pages 264–282. Springer Verlag, 2006.

[54] E. Deelman, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, and K. Vahi. Mapping Abstract Complex Workflows onto Grid Environments. *Journal of Grid Computing*, 1:25–39, 2003.

[55] C. Dumitrescu, D.H.J. Epema, J. Dünnweber, and S. Gorlatch. User Transparent Scheduling of Structured Parallel Applications in Grid Environments. In *HPC-GECO/CompFrame Workshop (held in conjunction with HPDC'06)*, 2006.

[56] C. Dumitrescu, I. Raicu, and I. Foster. DI-GRUBER: A Distributed Approach to Grid Resource Brokering. In *Proc. of the 2005 ACM/IEEE conference on Supercomputing*, pages 38–50, 2005.

[57] C. Dumitrescu, I. Raicu, and I. Foster. The design, usage, and performance of gruber: A grid usage service level agreement based brokering infrastructure. *Journal of Grid Computing*, 5(1):99–126, 2007.

[58] E. Elmroth and J. Tordsson. A Grid Resource Broker Supporting Advance Reservations and Benchmark-Based Resource Selection. In *Proc. of 7th International Workshop of Applied Parallel Computing, State of the Art in Scientific Computing*, pages 1061–1070, 2004.

[59] C. Ernemann, V. Hamscher, U. Schwiegelshohn, R. Yahyapour, and A. Streit. On Advantages of Grid Computing for Parallel Job Scheduling. In *Proc. of the 2nd IEEE/ACM International Symposium on Cluster Computing and the GRID (CCGrid2002)*, pages 39–46, 2002.

[60] C. Ernemann, V. Hamscher, A. Streit, and R. Yahyapour. Enhanced Algorithms for Multi-Site Scheduling. In *Proc. of the 3rd International Workshop on Grid Computing*, pages 219–231, 2002.

[61] H.E. Bal et al. The distributed ASCI Supercomputer Project. In *Operating Systems Review*, volume 34, pages 76–96, 2000.

[62] I. Foster et al. The Grid2003 Production Grid: Principles and Practice. In *Proc. of the 13th IEEE International Symposium on High Performance Distributed Computing (HPDC-13)*, 2004.

[63] I. Foster and C. Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128, 1997.

[64] I. Foster, C. Kesselman, C. Lee, B. Lindell, K. Nahrstedt, and A. Roy. A Distributed Resource Management Architecture that Supports Advance Reservation and Co-allocation. In *Proc. of 7th International Workshop on Quality of Service*, 1999.

[65] Ian Foster, Carl Kesselman, and Steven Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *International Journal Supercomputer Applications*, 15(3):200–222, 2001.

[66] J. Frey, T. Tannenbaum, I. Foster, M. Livny, and S. Tuecke. Condor-G: A Computation Management Agent for Multi-Institutional Grids. In *Proc. of the 10th IEEE Symposium on High Performance Distributed Computing (HPDC-10)*, pages 7–9, San Francisco, California, 2001.

[67] Andrew S. Grimshaw and William A. Wulf. Legion: The Next Logical Step Toward the World-wide Virtual Computer. *Communications of the ACM*, 40(1):39–45, 1997.

[68] E. Huedo, R. S. Montero, and I. M. Llorente. A Framework for Adaptive Execution in Grids. *Software: Practice and Experience*, 34:631–651, 2004.

[69] A. Iosup and D. H. J. Epema. GRENCHMARK: A Framework for Analyzing, Testing, and Comparing Grids. In *Proc. of the 6th IEEE/ACM International Symposium on Cluster Computing and the GRID (CCGrid2006)*, pages 313–320, 2006.

[70] G. V. Laszewski. Java CoG Kit Workflow Concepts for Scientific Experiments. Technical report, Globus Alliance, 2005.

[71] C. Lee and D. Talia. Grid Programming Models: Current Tools, Issues and Directions. In *Grid Computing: Making the Global Infrastructure a Reality*, pages 555–578. 2003.

[72] Y. C. Lee and A. Y. 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.

[73] David A. Lifka. The ANL/IBM SP Scheduling System. In *Proc. of the Workshop on Job Scheduling Strategies for Parallel Processing*, pages 295–303, London, UK, 1995. Springer-Verlag.

[74] J. Maassen. *Method Invocation Based Communication Models for Parallel Programming in Java*. PhD thesis, Vrije Universiteit, 2003.

[75] J. Maassen, T. Kielmann, and H. E. Bal. GMI: Flexible and Efficient Group Method Invocation for Parallel Programming. In *Proc. of the 6th Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers*, 2002.

[76] H.H Mohamed and D.H.J. Epema. Interfacing Different Application Types to the KOALA Grid Scheduler. *Future Generation Computer Systems*. Submitted.

[77] H.H Mohamed and D.H.J. Epema. KOALA: A Co-Allocating Grid Scheduler. *Concurrency and Computation: Practice and Experience*. Accepted for publication.

[78] H.H. Mohamed and D.H.J. Epema. An Evaluation of the Close-to-Files Processor and Data Co-Allocation Policy in Multiclusters. In *Proc. of IEEE International Conference on Cluster Computing 2004*, pages 287–298, 2004.

[79] H.H. Mohamed and D.H.J. Epema. The Design and Implementation of the KOALA Co-Allocating Grid Scheduler. In *Proc. of the European Grid Conference*, volume 3470 of *LNCS*, pages 640–650. 2005.

[80] S. Park and J. Kim. Chameleon: A Resource Scheduler in A Data Grid Environment. In *Proc. of the 3rd IEEE/ACM International Symposium on Cluster Computing and the GRID (CCGrid2003)*, pages 256–263, 2003.

[81] A. Pouwelse, P. Garbacki, J. Wang, A. Bakker, J. Yang, A. Iosup, D.H.J. Epema, M. Reinders, M.R. van Steen, and H.J. Sips. Tribler: A social-based peer-to-peer system. *Concurrency and Computation: Practice and Experience*. to appear.

[82] Diego Puppin, Stefano Moncelli, Ranieri Baraglia, Nicola Tonellotto, and Fabrizio Silvestri. A Grid Information Service Based on Peer-to-Peer. In *Proc. of the 11th International Euro-Par Conference*, 2005.

[83] R. Raman, M. Livny, and M. Solomon. Policy Driven Heterogeneous Resource Co-Allocation with Gangmatching. In *Proc of the 12th IEEE International Symposium on High Performance Distributed Computing (HPDC-12)*, pages 80–89, 2003.

[84] K. Ranganathan and I. Foster. Decoupling Computation and Data Scheduling in Distributed Data-Intensive Applications. In *Proc. of the 11th IEEE International Symposium on High Performance Distributed Computing (HPDC-11)*, pages 352–358, 2002.

[85] R.Buyya, M. Murshed, D. Abramson, and S. Venugopal. Scheduling Parameter Sweep Applications on Global Grids: a Deadline and Budget Constrained Costtime Optimization Algorithm. *Software: Practice and Experience*, 35:491–512, 2005.

[86] T. Röblitz and A. Reinefeld. Co-reservation with the concept of virtual resources. In *Proc. of the 5th IEEE/ACM International Symposium on Cluster Computing and the GRID (CCGrid2005)*, pages 398–406, 2005.

[87] T. Röblitz, F. Schintke, and A. Reinefeld. Resource Reservations with Fuzzy Requests. *Concurrency and Computation: Practice and Experience*, 18(13):1681–1703, 2006.

[88] J. Roozenburg. Secure Decentralized Swarm Discovery in Tribler. Master's thesis, Delft University of Technology, 2006.

[89] K. Seymour, H. Nakada, S. Matsuoka, J. Dongarra, C. A. Lee, and H. Casanova. Overview of GridRPC: A Remote Procedure Call API for Grid Computing. In *Proc. of the 5th International Workshop on Grid Computing (GRID 2002)*, pages 274–278, 2002.

[90] W. Smith, I. Foster, and V. Taylor. Scheduling with Advanced Reservations. In *Proc. of International Parallel and Distributed Processing Symposium (IPDPS)*, pages 127–132, 2000.

[91] O. Sonmez, H. Mohamed, and D. Epema. Communication-Aware Job Placement Policies for the KOALA Grid Scheduler. In *Proc. of the Second IEEE International Conference on e-Science and Grid Computing*, 2006.

[92] A. Streit, D. Erwin, T. Lippert, D. Mallmann, R. Menday, M. Rambadt, M. Riedel, M. Romberg, B. Schuller, and Ph. Wieder. UNICORE - From Project Results to Production Grids. In *Grid Computing: The New Frontiers of High Performance Processing, Advances in Parallel Computing*, pages 357–376, 2005.

[93] D. Thain, J. Basney, S. Son, and M. Livny. The Kangaroo Approach to Data movement on the Grid. In *Proc. of the 10th IEEE Symposium on High Performance Distributed Computing*, pages 325–333, 2001.

[94] D. Thain, J. Bent, A. Arpaci-Dusseau, R. Arpaci-Dusseau, and M. Livny. Gathering at the Well: Creating Communities for Grid I/O. In *Proc. of the IEEE/ACM conference on Supercomputing* , pages 58–68, 2001.

[95] D. Thain, T. Tannenbaum, and M. Livny. Distributed Computing in Practice: the Condor Experience. *Concurrency and Computation: Practice and Experience*, 17(2-4):323–356, 2005.

[96] R. V. van Nieuwpoort, J. Maassen, T. Kielmann, and H. E. Bal. Satin: Simple and Efficient Java-based Grid Programming. *Scalable Computing: Practice and Experience*, 6:19–32, 2005.

[97] R. V. van Nieuwpoort, J. Maassen, G. Wrzesinska, R. Hofman, C. Jacobs, T. Kielmann, and H. E. Bal. Ibis: a Flexible and Efficient Java based Grid Programming

Environment. *Concurrency and Computation: Practice and Experience*, 17:1079–1107, 2005.

[98] Rob V. van Nieuwpoort, Jason Maassen, Rutger Hofman, Thilo Kielmann, and Henri E. Bal. Ibis: an Efficient Java-based Grid Programming Environment. In *Proc. of the ACM JavaGrande ISCOPE 2002 Conference*, pages 18–27, 2002.

[99] R.V. van Nieuwpoort, J. Maassen, H.E. Bal, T. Kielmann, and R. Veldema. Wide-Area Parallel Programming Using the Remote Method Invocation Model. *Concurrency and Computation: Practice and Experience*, 12(8):643–666, 2000.

[100] S. Venugopal, R. Buyya, and L. Winton. A Grid Service Broker for Scheduling e-Science Applications on Global Data Grids: Research Articles. *Concurrency and Computation: Practice and Experience*, 18(6), 2006.

[101] J. Voeckler. Yet Another Concrete Planner. In *Virtual Data Workshop*, 2004.

[102] W.Cirne, D. Paranhos, L. Costa, E. Santos-Neto, F. Brasileiro, J. Sauve, F. A. B. Silva, C. O. Barros, and C. Silveira. Running Bag-of-Tasks Applications on Computational Grids: The MyGrid Approach. In *Proc. of the International Conference on Parallel Processing*, pages 407–416, 2003.

[103] J. B. Weissman and A. S. Grimshaw. A Federated Model for Scheduling in Wide-Area Systems. In *Proc. of the 5th IEEE International Symposium on High Performance Distributed Computing (HPDC-5)*, pages 542–550, 1996.

[104] J. B. Weissman and X. Zhao. Scheduling Parallel Applications in Distributed Networks. *Cluster Computing*, 1:109–118, 1998.

[105] C. Weng and X. Lu. Heuristic Scheduling for Bag-of-Tasks Applications in Combination with QoS in the Computational Grid. *Future Generation Computer Systems*, 21:271–280, 2005.

[106] W. Xiaohui, D. Zhaohui, Y. Shutao, H. Chang, and L. Huizhen. CSF4: A WSRF Compliant Meta-Scheduler. In *Proc. of World Congress in Computer Science Computer Engineering, and Applied Computing*, pages 61–67, June 2006.

[107] C. S. Yeo and R. Buyya. A Taxonomy of Market-based Resource Management Systems for Utility-driven Cluster Computing. *Software: Practice and Experience*, 36:1381–1419, 2006.

[108] J. Yu and R. Buyya. A Taxonomy of Scientific Workflow Systems for Grid Computing. *SIGMOD Rec.*, 34:44–49, 2005.

# Acknowledgments

I would like to thank my supervisor Henk Sips, and my co-supervisor Dick Epema for their contribution to this thesis and general support during my PhD in Delft. I would also like to thank Henri Bal for his inspiring talks in grids that I was able to attend. Part of the work for this thesis was carried out in the context of the Virtual Laboratory for e-Science project (www.vl-e.nl), which is supported by a BSIK grant from the Dutch Ministry of Education, Culture and Science (OC&W), and which is part of the ICT innovation program of the Dutch Ministry of Economic Affairs (EZ). I am very grateful for this support. Special thanks to Franca Post of CICAT, which is the central international liaison office at TU Delft, for her support on administrative issues since I arrived in the Netherlands in 1999.

Many thanks to Paulo Anita and Kees Verstoep for solving all my DAS-related problems that required system adminstrator's attention. My colleagues in the Parallel and Distributed Systems Group, Alexandru Iosup, Ozan Sonmez, Catalin Dumitrescu, Mathieu Jan, Jeremy Buisson and I have had many brainstorming sessions during our grid meetings; thanks guys! I truly appreciate your inputs. Finally, I would also like thank Wouter Lammers, who contributed a lot to designing and implementing KOALA's runners framework.

# Summary

Grid computing is an emerging form of distributed computing, distinguished from traditional forms by its focus on large-scale, multi-organizational resource sharing and innovative applications. Like any computing infrastructure, the grid infrastructure is made up of hardware and software, which have been advancing rapidly. This advance is facilitated greatly by innovations in building fast and powerful commodity computers and networks, which has been accompanied by a drop in their prices. Also, the software technology for building key components of the grid software infrastructure has become easier to use and more robust. These key components include software for building single-cluster and multicluster systems, and grid middleware. Multicluster systems are formed by joining multiple, geographically distributed clusters interconnected by high-speed wide-area networks. Grid middleware offers transparent access to a wide variety of distributed grid resources to users. Through the use of the grid middleware by means of the simple interfaces it provides, a normal user does not have to know the technical details on how to access these resources. The advance of the grid infrastructure and what it is promising to offer has resulted in new grid applications and grid application types that are attempting to take advantage of the grid.

Grids need high-level schedulers that can be used to manage resources across multiple organizations; we call such schedulers Grid Resource Management Systems (GRMS). GRMSs do not actually own the resources of grids, and neither do they have full control over the jobs that are running in grids. This makes resource management by GRMSs very difficult. An important possible requirement to a GRMS is to support *co-allocation*, i.e., the simultaneous or coordinated access of single applications to resources of possibly multiple types in multiple locations. Co-allocation presents new challenges to resource management in grids. These challenges include allocating resources in multiple sites, guaranteeing the simultaneous availability of the co-allocated resources when they are about to be accessed by the applications, and managing sets of highly dynamic grid resources. To deploy jobs on the resources they have been allocated by a grid scheduler, good job deployment mechanisms are required. The emergence of new applications types which have unique characteristics and of the grid infrastructure itself poses the challenge of deploying jobs in grids.

In this thesis we address the challenges of co-allocation and of grid application deployment. For this purpose we have designed and implemented the KOALA GRMS, which has been deployed on the second and third generations of the Distributed ASCI Supercomputer (DAS). The DAS is a multicluster system consisting of five geographically distributed clusters interconnected by a high speed network in four universities in the Netherlands. KOALA has proven to be working reliably on the DAS testbed with over 500,000 jobs already submitted successfully with it.

In Chapter 1 of this thesis we introduce the problem of grid resource management, and in particular, we state the challenges of co-allocation and application deployment in grids that we address in this thesis. In Chapter 2 we describe in detail the background on grids that is required to read this thesis. In addition, we present our model for resource management and co-allocation in grids.

In Chapter 3 we describe the design of the KOALA GRMS. The architecture of KOALA consists of three major layers, namely, the scheduler, the runners framework, and the runners, which are job submission and monitoring tools for specific application types. The scheduler is equipped with placement policies that are used to place jobs on suitable execution sites, and with a claiming policy that is used to claim processors for jobs at their scheduled start times. The runners framework hides the heterogeneity of the grid by providing a set of functions to the runners for commonly used grid job submission operations. The runners framework simplifies the development of runners and therefore, it encourages the addition of runners for different application types.

In Chapter 4 the KOALA job policies are discussed. These job policies are the Close-to-Files (CF) and the Worst-Fit (WF) policies, which are placement policies, and the Incremental Claiming Policy (ICP), which is a claiming policy. The placement policies address the co-allocation challenge of the simultaneous allocation of resources in multiple sites to single jobs. The CF policy has been designed to minimize the delays when starting a job caused by long input file transfers, and the WF policy has been designed to balance processor usage among the clusters and if possible to minimize file transfer times as well. The claiming policy addresses the co-allocation challenge of guaranteeing the simultaneous availability of the processors in the absence of advance processor reservations. The ICP policy tries to make processors available for jobs, if needed by finding processors at other sites than where they were allocated or, if permitted, by forcing processor availability through preemption of running jobs.

Chapters 5 and 6 present the results of the experiments we have conducted in the DAS to assess the operation of the KOALA job policies and the KOALA runners. The results show that indeed a reliable co-allocating grid scheduler can actually be designed, implemented, and deployed in a multicluster system, that co-allocation is a useful mechanism to have in a GRMS, and that enabling grid applications is simplified by separating scheduling from application deployment. We have also learned that grid resources are

hardly reliable, which was evident even in a homogeneous testbed like DAS-2, which is centrally managed. Therefore, designing reliable mechanisms and studying them through experimentation in real environments, is very important.

# Samenvatting

Grid computing is een opkomende vorm van distributed computing die zich van traditionele vormen onderscheidt door een focus op het grootschalige, multi-organisationele, gemeenschappelijke gebruik van *resources* en op innovatieve applicaties. Zoals iedere computerinfrastructuur, bestaat de grid-infrastructuur uit hardware en software, en in beide worden grote vorderingen gemaakt. Deze vorderingen worden mogelijk gemaakt door innovaties in het bouwen van snelle en krachtige standaard-computers en standaard-netwerken, hetgeen gepaard gaat met grote prijsdalingen. Ook wordt de software-technologie voor het bouwen van de sleutelcomponenten van de software-infrastructuur van grids steeds gemakkelijker in het gebruik en steeds robuuster. Deze sleutelcomponenten omvatten software voor het bouwen van multiclustersystemen en grid middleware. Multiclustersystemen worden gevormd door het samenvoegen van meerdere geografisch gespreide clusters die verbonden worden door snelle *wide-area* netwerken. Grid middleware biedt transparante toegang tot een verscheidenheid aan gedistribueerde grid-*resources* aan de gebruikers. Door het gebruik van grid middleware via de eenvoudige interfaces die het biedt, hoeft een gewone gebruiker geen kennis te hebben van de technische details van de toegang tot de *resources*. De vooruitgang van de grid-infrastructuur heeft geresulteerd in nieuwe (typen van) grid-applicaties die proberen de mogelijkheden van grids uit te buiten.

Grids hebben hoog-niveau schedulers nodig die gebruikt kunnen worden om *resources* te beheren die in het bezit zijn van meerdere organisaties; zulke schedulers worden wel Grid Resource Management Systemen (GRMS) genoemd. Een GRMS bezit de *resources* niet werkelijk zelf, en ook heeft zo'n systeem niet de volledige controle over de jobs die in een grid worden uitgevoerd. Dit maakt het beheer van *resources* door een GRMS erg moeilijk. Een belangrijke mogelijke eis aan een GRMS is om *co-allocatie* te ondersteunen, d.w.z. de gelijktijdige of gecoördineerde toegang van een enkele applicatie tot *resources* van mogelijkerwijs meerdere types op meerdere locaties. Co-allocatie brengt nieuwe uitdagingen voor het beheer van *resources* in grids met zich mee. Deze uitdagingen zijn onder meer het toewijzen van *resources* op meerdere locaties, het garanderen van de gelijktijdige beschikbaarheid van de *resources* die met co-allocatie toegewezen zijn op het moment dat een applicatie ze wil gaan gebruiken, en het beheren van de zeer

dynamische grid *resources*. Voor het activeren van jobs op de *resources* die door een grid scheduler zijn toegewezen zijn goede mechanismen vereist. Het ontstaan van nieuwe typen applicaties met ieder hun eigen karakteristieken en van de grid-infrastructuur zelf leidt tot de uitdaging van het activeren van jobs in grids.

In dit proefschrift gaan we de uitdagingen aan van co-allocatie en van het activeren van jobs in grids. Hiertoe hebben we het KOALA GRMS ontworpen en geïmplementeerd, dat geïnstalleerd is op de tweede en derde generatie van de Distributed ASCI Supercomputer (DAS). De DAS is een multiclustersysteem dat bestaat uit vijf geografisch gespreide clusters verbonden door een snel netwerk bij vier universiteiten in Nederland. KOALA heeft bewezen dat het betrouwbaar functioneert op de DAS; tot nu toe zijn al meer dan 500.000 jobs via KOALA aan de DAS aangeboden.

In Hoofdstuk 1 van dit proefschrift introduceren we het probleem van het beheer van grid *resources*, en in het bijzonder formuleren we de uitdagingen van co-allocatie en van het activeren van applicaties in grids die we in dit proefschrift aangaan. In Hoofdstuk 2 beschrijven we in detail het benodigde achtergrondmateriaal om dit proefschrift te lezen.

In Hoofdstuk 3 presenteren we het ontwerp van het KOALA GRMS. De architectuur van KOALA bestaat uit drie lagen, nl. de scheduler, het *runners framework*, en de *runners*; deze laatste zijn software-componenten voor het aanbieden en het volgen van de executie van specifieke typen applicaties. De scheduler is uitgerust met *placement policies* voor het toewijzen van geschikte *execution sites*, en van een *claiming policy* voor het opeisen van processoren ten behoeve van jobs als deze daadwerkelijk willen starten. Het *runners framework* schermt de heterogeniteit van het grid af door een verzameling functies aan de *runners* ter beschikking te stellen voor veel voorkomende operaties voor het activeren van grid jobs. Het *runners framework* vereenvoudigt de ontwikkeling van *runners*, en stimuleert zodoende de toevoeging van *runners* voor verschillende typen applicaties.

In Hoofdstuk 4 worden de *job policies* van KOALA besproken; deze zijn de *placement policies Close-to-Files* (*CF*) en *Worst-Fit* (*WF*), en de *Incremental Claiming Policy* (*ICP*) voor het opeisen van toegewezen processoren. De *placement policies* vormen een antwoord op de uitdaging van de voor co-allocatie vereiste gelijktijdige toewijzing van *resources* in meerdere locaties aan een enkele job. De *policy* CF is ontworpen om de vertraging in het starten van een job veroorzaakt door het overbrengen van invoerbestanden te minimaliseren, en de *policy* WF is ontworpen om het processorgebruik tussen de clusters in balans te houden en indien mogelijk ook de tijd nodig voor het overbrengen van bestanden te minimaliseren. De *claiming policy* vormt een antwoord op de uitdaging van de voor co-allocatie vereiste garantie van de gelijktijdige beschikbaarheid van processoren als er geen mechanismen aanwezig zijn om processoren te reserveren. De *policy* ICP probeert processoren beschikbaar te maken voor jobs, indien nodig door processoren te vinden op andere locaties dan waar deze oorspronkelijk waren toegewezen of, indien toegestaan, door de beschikbaarheid van processoren af te dwingen via de pre-emptie van

draaiende jobs.

Hoofdstukken 5 en 6 presenteren de resultaten van de experimenten die we in de DAS hebben uitgevoerd om de werking te beoordelen van de *job policies* en de *runners* van KOALA. De resultaten tonen aan dat het inderdaad mogelijk is om een betrouwbare grid scheduler die co-allocatie ondersteunt te ontwerpen, te implementeren, te installeren, en te gebruiken in een multiclustersysteem, dat co-allocatie een nuttig mechanisme is in een GRMS, en dat het executeren van grid-applicaties vereenvoudigd wordt door de scheduling van deze applicaties te scheiden van het activeren ervan. We hebben ook ondervonden dat grid resources niet erg betrouwbaar zijn, hetgeen zelfs het geval is in de DAS, die centraal beheerd wordt. Derhalve is het erg belangrijk om betrouwbare mechanismen te ontwerpen en met behulp van experimenten in een werkelijke omgeving te bestuderen.

# Curriculum Vitae

Hashim Mohamed was born in Arusha, Tanzania, on October 11th, 1973. In June 1995, he joined the University of Dar-es-salaam in Tanzania for a B.Sc. degree in computer science and he graduated in 1998. Between June 1998 and January 1999 he worked at the University of Dar-es-salaam computing center as a systems analyst/programmer. In February 1999, he moved to Delft University of Technology in the Netherlands for a master of science program and he graduated in 2001 with an M.Sc. in Technical Informatics. In 2003, he joined the Parallel and Distributed Systems Group of Delft University of Technology as a PhD student. His research interests are in the areas of distributed systems, multi-cluster systems, and grids in general. In addition to research, his other interests include installing, configuring, and managing clusters of computers.