Adding support for new application types to the Koala grid scheduler

Wouter Lammers

October 2005

Delft University of Technology

Adding support for new application types to the Koala grid scheduler

Master's Thesis Computer Science

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

Wouter Lammers

October 2005

Author

Wouter Lammers

Title

Adding support for new application types to the Koala grid scheduler

Date

November 7, 2005

Graduation Committee

Prof. dr. ir. H.J. Sips (chair)	Delft University of Technology Parallel and Distributed Systems group
Ir. dr. D.H.J. Epema	Delft University of Technology Parallel and Distributed Systems group
Ir. H.H. Mohamed	Delft University of Technology Parallel and Distributed Systems group
Ir. F.H. Post	Delft University of Technology Computer Graphics & CAD/CAM group

Abstract

Different application types require different procedures to execute on a grid, for example, procedures to setup wide area communication or to initialise specialised hardware. The intricacies of these procedures can hinder the effective use of a grid. At the Parallel and Distributed Systems group at the TU Delft, we are developing the KOALA grid scheduler. KOALA's modular structure allows us to incorporate different submission procedures for different application types through its so-called runners. We have designed and implemented four of these runners for KOALA. The first one, the KRunner, is used to run relatively simple applications. The GRunner and the IRunner are designed to run applications which use Ibis, a special Java communication library optimised for wide area network connections. The last runner, the MRunner, is used to run malleable jobs, which are jobs that can change the number of grid resources they occupy during runtime. In addition, all runners support co-allocation, i.e., the simultaneous allocation of resources such as processors and input files in multiple clusters. We have run many experiments which show the ease of use and the reliability of the runners.

Preface

This thesis describes the work I have done for my graduation from Delft University of Technology at the Parallel and Distributed Systems group. It presents the design and implementation of the work I did for the KOALA co-allocating grid scheduler. I would like to thank Almudena, my girlfriend, for her continuing support throughout these last nine months. I would also like to thank my parents for their support, especially for the last few months. Thanks to my friends in the 'fishbowl', especially Renze, for letting me vent my thoughts during coffee breaks. I would like to thank my supervisor Dick Epema for his guidance throughout this project and for improving my writing skills on many fronts. And last, but certainly not least, I would like to thank Hashim Mohamed for helping me through the fickle environment of grid computing and Alexandru Iosup and Hashim for our many productive brainstorm sessions.

Wouter Lammers
Delft, October 2005

Contents

Chapter 1

Introduction

In the scientific world there are many problems which need more processing power than any single computer can provide, such as protein folding in chemistry, seismic applications for soil research, but also computing climate predictions, modelling sediments in a riverbank, and finite element computations for large bridges and buildings. To solve these problems, scientists employ supercomputers or computing clusters, which are usually expensive pieces of hardware. To make better use of these resources, they can be connected to form so-called computational grids, which can span multiple organisations. Connecting such heterogeneous computing resources with their own specific access rights and technical intricacies produces many difficulties. In addition to running large batches of sequential jobs, also co-allocated applications, which are large parallel programs with parts executing at different physical locations, can be run on a grid. Different application types require different submission procedures to run on a grid. For example, these different submission procedures can be necessary to setup wide area communication, to initialise special grid hardware, or to implement a startup barrier for multi-component applications.

The KOALA co-allocating grid scheduler is developed in the Parallel and Distributed Systems (PDS) group of Delft University of Technology to facilitate easy and reliable submission of jobs to a grid. With KOALA, users do not have to figure out themselves where their program will run in a grid and KOALA manages the execution of the programs. KOALA is designed to overcome many problems which are frequently encountered in grids. Programs executed on a grid often need special startup procedures to make use of specialised hardware and/or software. KOALA supports the submission of different job types through its so-called runners. A runner is responsible for the submission of a particular application type to a grid. The first version of KOALA has been released on our testbed, the Distributed ASCI Supercomputer (DAS) [3]. We have extensive experience with running jobs on the DAS using KOALA.

We have implemented large parts of the basic runner architecture and the runner communication protocol. In addition we have created four runners. The KRunner is the default runner used to run jobs which do not require special procedures. The KRunner only allocates processors and starts job components without any support for communication. It transfers the necessary input files before a job starts executing and retrieves all output files a job produces.

The GRunner and the IRunner are designed to run Ibis programs, which are programs that use a specialised Java communication library. Ibis has been developed at the Vrije Universiteit (VU) in Amsterdam and its goal is to design and implement an efficient and flexible Java-based programming environment for grid computing. Included in the Ibis distribution there is a submission tool called grun. This tool facilitates running Ibis jobs but does not support co-allocation or fault tolerance. The GRunner combines the co-allocation functionality and fault tolerance of KOALA with the easy submission capabilities of grun.

The IRunner is an alternative implementation for running Ibis jobs with KOALA. It is based on the KRunner and does not submit through grun. It does support the file management functionalities which are implemented in the KRunner.

The last runner is called the MRunner and is designed to support the submission of malleable jobs, which are jobs that can change their size at runtime. Not many implementations of grid schedulers exist which can handle malleable jobs. KOALA's modular design allows us to construct the MRunner and support malleable jobs in KOALA. We have also implemented additional scheduling policies in KOALA to handle these jobs. We present our design and implementation of these runners as well as experiments which show their reliability.

The rest of this thesis is organised as follows. In Chapter 2 we discuss the background information of our work. KOALA and the basic runner structure are detailed in Chapter 3. The runner for Ibis jobs is explained in Chapter 4, followed by details about the runner for malleable jobs in Chapter 5. We conclude the thesis in Chapter 6 with our conclusions and suggestions for further research.

Chapter 2

Background

In this chapter we will discuss the background information of our work. We start with a short introduction to grid computing in Section 2.1. In Section 2.2 we define a general grid model that we use throughout this report. We also discuss our testbed, the Distributed ASCI Supercomputer (DAS-2) in Section 2.3. We introduce our grid scheduler, called KOALA, in Section 2.4, followed by a short introduction to the GRENCHMARK benchmarking tool in Section 2.5.

2.1 Introduction to grid systems

Science knows many intractable problems for which no easy solution exists and for which brute force solutions would take too much time (months, even years). Examples of these problems [21] are the design verification of microprocessors, molecular simulations of liquid crystals, ground-penetrating radar, hard cryptographic problems, weather (climate) forecasting, protein folding algorithms, and many more. In this section we will discuss grid systems in general and the problems they introduce.

2.1.1 Motivation for a grid system

A logical step in solving above mentioned kind of problems is to construct parallel supercomputers to get more processing power than a single processor system offers. Another approach is using many networked single processor computers and combine them into a cluster as is done in Condor pools [14, 43]. These clusters and supercomputers have been around for quite some time, but the next step, grid computing, is still relatively new. The same idea as has been applied to single processor systems has been applied to existing clusters and supercomputers: ‘more is better’. In a grid system, multiple clusters and/or supercomputers are connected and used as a transparent (very large) computing entity. Although grid computing is receiving increasing attention from researchers, no lasting standard has emerged yet. The main difference between grid computing and parallel computing is the absence of a centralised resource manager in grids.

2.1.2 Resources in a grid system

Grids can be comprised of many different computing entities like a supercomputer or desktop cluster, as well as data storage entities like large database repositories. Some grids even include specialised equipment which are only present in remote laboratories, e.g., access to a large telescope. In this report we will focus on using grid systems for solving large computations. The components of a grid will be referred to as computation sites, or just sites. A grid is called homogeneous when all its sites operate on the same hardware and software. A heterogeneous grid can have different configuration at every site, e.g., processor speed, memory capacity, interconnection bandwidth and latency, as well as different operating systems and grid middleware. The resources of a grid are the components a site provides to users of that grid. Each site of a grid system has its own autonomous local resource manager.

2.1.3 Difficulties in a grid system

Many difficulties arise when connecting many different systems together into a grid system. We find there are five main issues when constructing a grid:

1. Heterogeneity

By definition, a grid system combines different resources. All these resources have their own methods for accessing and controlling them. In order to use all these different resources, a common interface has to be devised to make these resources interoperable. Sometimes the software which makes this possible is called grid middleware.

2. Resource discovery

When a grid encompasses hundreds of sites it is not enough to have a static configuration of available resources. A dynamic way of finding available resources has to be made as well as flexible mechanisms for resources to enter and leave the grid at any time.

3. Scheduling

Because all sites in a grid are autonomous there is no central scheduling component in grid systems, at least not an enforcing central scheduler. Grid schedulers have to work with outdated information and are left to the mercy of the site schedulers (local schedulers) whether or not their job will in fact be submitted to that site.

4. Communication

Sites normally have homogeneous intra-connections, but are connected themselves to the grid in many different ways, e.g., behind a restrictive firewall, low bandwidth connections, etc. Although TCP/IP enables successful communication on the Internet with these difficulties, TCP/IP may not be suitable for the high speed computations that are run on grid systems.

5. Security

Many different organisations can be involved in the construction of a grid. Each having different access rights to the resources of the grid. Constructing a uniform security model for the many different components of a grid is not a trivial task.

The central theme of this report will be centred around co-allocation in grids. By co-allocation we mean both the scheduling of jobs and claiming of required resources in multiple sites. All of the above difficulties are, in varying degrees, encountered when doing co-allocation. Because of these many difficulties grids tend to become 'fickle'. There are many possibilities of (unforeseen) failures leading to the need for robust and fault tolerant techniques when dealing with a grid. For a more detailed description of resource management in grid systems see our previous work [32].

2.2 A grid model for resource management

In the literature on grid related subjects, names of techniques and concepts tend to be (ab)used for different meanings. In this section we will define some concepts to avoid misunderstandings. Our general grid model consists of jobs and sites. Jobs are programs that are to be executed on one or more sites. Sites contain the resources a job can request. These resources could be a number of processors, some bandwidth, storage space or even a software licence [43]. In our model we are mainly concerned with computing, and thus by resources we mean a number of processors.

Sites are groups of resources located remotely from each other. All sites have autonomous local resource managers. This means that jobs are scheduled, monitored, and controlled on a site independently from other sites. Our model contains a global grid scheduler. We recognise the bottleneck in such an approach and future work will include a distributed global scheduler. Jobs are submitted to this global scheduler, which then has to cooperate with the local schedulers which will actually execute the job.

2.2.1 Job types

Jobs can require a different number of processors leading to different job types. In grid systems a job can be executed at multiple locations that we call sites. We divide a job into a set of components, each requiring their own number of processors. The components themselves have to be executed at a single site, but different components can run at different sites.

In the first part of this thesis we will consider only *rigid* jobs, which are jobs with a fixed number of processors. The size of the components is known in advance and does not change during the running time of the job. Next to *rigid* jobs there are *malleable* or *evolving* jobs [19]. We will discuss these job types in Chapter 5. We can furthermore divide *rigid* jobs into the following categories:

- For **fixed** jobs the size of each component is predefined and the execution site for each component is fixed. A fixed job of only one component is sometimes called a **total** job.

- **Non-fixed** jobs also have preset component sizes but the components can run on any site.
- **Semi-fixed** have some components with a fixed site but others are free to be executed anywhere.
- For a **flexible** job the component sizes are not known in advance. They can be configured in any way that meets resource availability.

2.2.2 Sites

We define an *execution site* as the site where a job component is executed. Before a job can run, its execution sites have to have the necessary files present. These files are obviously the executable itself, but also, possibly very large, input files. The site which hosts the input data is called the *file site*. The file site does not necessarily have to be the site where the user submits his job request, i.e., the *submission*[1] *site*.

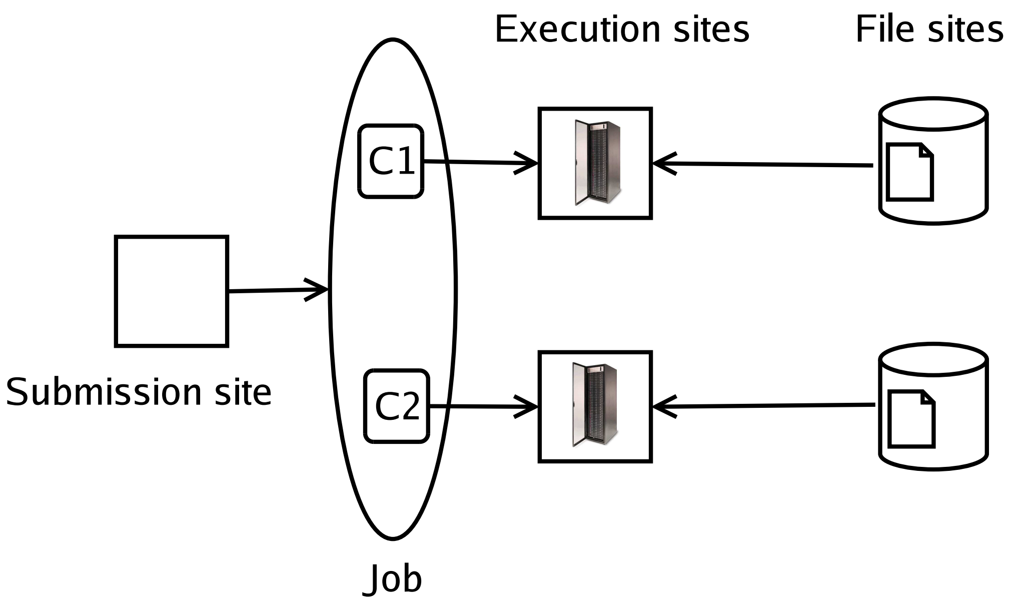


Figure 2.1: Different sites are used when running a job.

2.2.3 The job life cycle

Terms like scheduling and allocation seem to be used interchangeably in the literature. In our model *scheduling* means finding a suitable site for a job to run on, or more generally, to select when and which resources will be appointed to a job request. To avoid confusion we will call this *placing* instead of *scheduling*. The actual *claiming* of these resources is done in a later phase. *Claiming* means that actual access to the resource is available. *Placing* **and** *claiming* a job is what we mean by *allocation*.

[1]Although we sympathise with Steve Hortney's campaign, we will use the term *submission* instead of *submittal*, since jobs **are** at the mercy of the scheduler.

Allocation done for a single job which can run on multiple sites is what we define as *co-allocation* (which thus includes *co-scheduling*). From now on we will refer to our global scheduler as the co-allocator, since that is what it is in fact doing according to our definition.

From its inception (submission) to termination (either successfully or erroneously) a job goes through multiple phases. Of course there are the ‘allocating’ and ‘executing’ main phases, but from a co-allocating perspective we are only interested in the allocating part of the job life cycle and to some extent the terminating part. The job life cycle consists of the following phases:

1. Submission: the job is submitted to the grid.
2. Placement: the job is waiting for a suitable execution site.
3. Staging: input files are transferred, job environment is set up. (This is an optional phase.)
4. Claiming: the job is started on its execution site.
5. Running: the job is executing.
6. Finishing: the job is removed from the execution site.
7. Staging: output files are transferred. (This is an optional phase.)

After submission, job components are placed on available sites. Placement does not mean the resources have already been claimed. After the placement phase, and the optional staging phase, the job tries to claim the resources for its components. When all the components are claimed, the job starts running and ends in failure or success. A final staging phase retrieves the output of the job.

2.3 The DAS-2 testbed

The Distributed ASCI Supercomputer (DAS-2) is a wide-area distributed computer of 200 Dual Pentium-III nodes [3]. The DAS-2 consists of five clusters of nodes located at five Dutch universities; the Vrije Universiteit Amsterdam (VU), the University of Amsterdam (UVA), Delft University of Technology (DUT), the University of Leiden (UL) and the University of Utrecht (UU).

Access to nodes is exclusive, which means we are basically only scheduling using space sharing and not time sharing. The local resource manager currently is Sun Grid Engine (SGE). At the time of the first experiments with KOALA, the local resource manager was an OpenPBS/Maui combination [5]. Because of slow performance and reliability issues OpenPBS was replaced. Each site has 32 nodes except for the site of the VU, which has 72 nodes. The nodes are connected through Myrinet, which is a fast LAN (1200 mbit/s), and Fast Ethernet for file transfers. The sites are interconnected through the Dutch university Internet backbone. Each node has 2 Pentium III processors running at 1 Ghz, at least 1 GB of RAM and at least a 20 GB local IDE disk.

All the nodes have the same configuration and are running RedHat Linux. The DAS-2 is a homogeneous system and could therefore be considered a to be more of a multicluster than a grid. However, for scheduling and communication research the DAS-2 does exhibit the typical behaviour of a grid system. The Globus Toolkit 3.2 [4, 22] is used to run jobs on multiple sites simultaneously. Every cluster has one fileserver, all the other nodes are compute nodes and mount the fileserver in their respective cluster, using NFS. Figures 2.2 and 2.3 show a global overview of the DAS-2 multicluster.

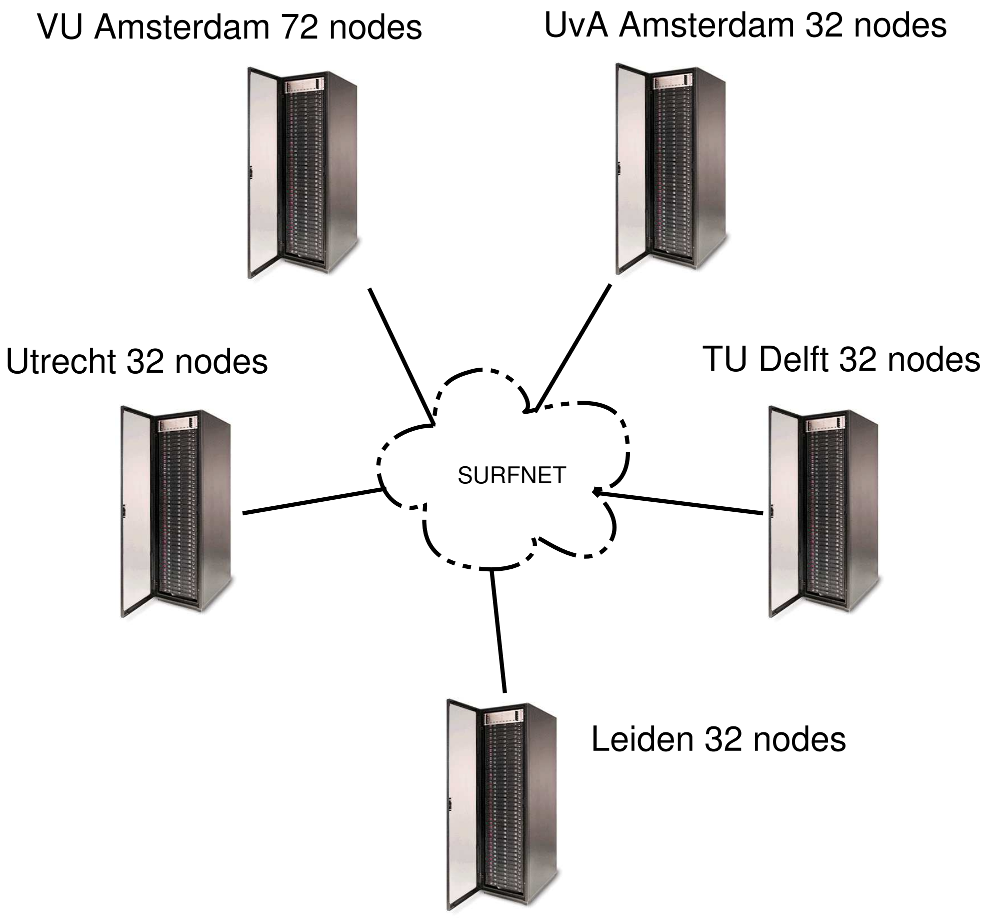


Figure 2.2: The DAS-2 consists of 5 clusters.

Every cluster has an autonomous instance of SGE running. Jobs can be directly submitted to the nodes of a cluster using this local resource manager. The SGE scheduler supports some form of backfilling. This means that whenever there is a large job in the pending queue for which not enough nodes are available, a smaller job which can be accommodated will ‘jump the queue’. It will be executed before the large job.

2.4 KOALA

KOALA is a grid scheduler that has been designed, implemented, and deployed on

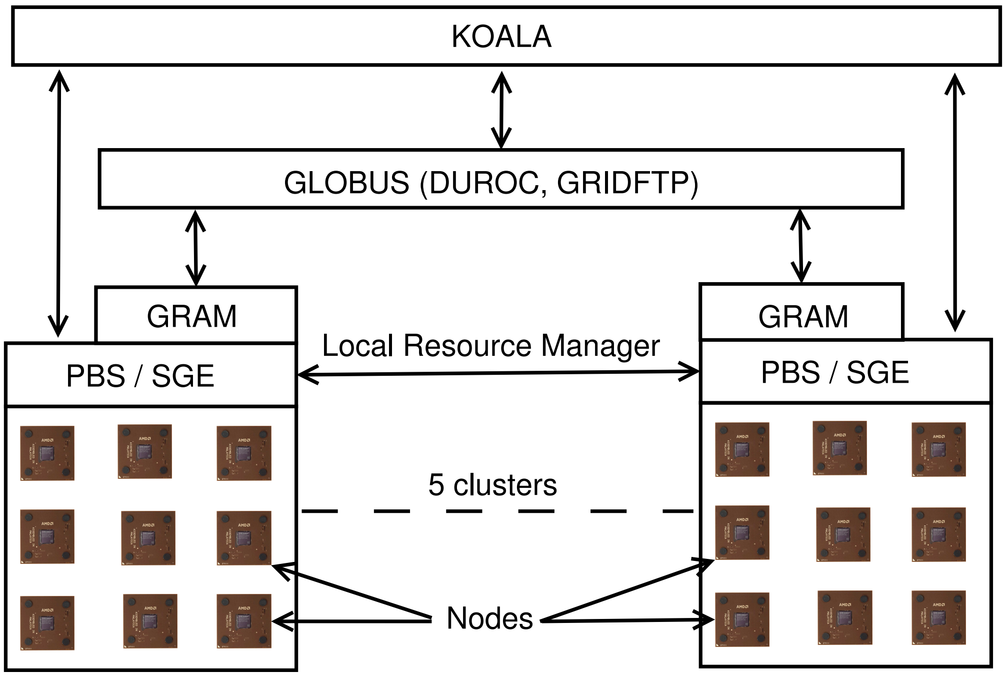


Figure 2.3: Overview of global and local software components in the DAS-2.

the DAS-2 multicluster system in the context of the Virtual Lab for e-Science (VL-e) research project. This report mainly discusses the parts of KOALA responsible for submitting applications to a grid, i.e., the runners. The runners also interface with the part of KOALA that finds suitable locations for the jobs, the co-allocator. KOALA has been designed for jobs which need co-allocation and subsequently supports processor and data co-allocation. KOALA interfaces to the local SGE schedulers using the Globus toolkit. It can submit co-allocated MPI and Ibis jobs; Ibis is a communication library written in Java, developed at the Vrije Universiteit Amsterdam [2, 52]. The main strength of KOALA is its ability to submit non-fixed jobs. This means users do not need to specify explicit clusters where their components have to run. KOALA will try to schedule the job as efficient as possible using its Close-to-Files policy. This is very useful for large co-allocated jobs which would otherwise be impossible to run if a system is heavily loaded. Even for smaller, one component jobs, KOALA can ease the submission to a grid. We will discuss KOALA in more detail in Chapter 3.

2.5 GRENCHMARK

GRENCHMARK is a synthetic grid workload generator and submitter. It can generate workloads for a grid consisting of many types of applications in multiple configurations. GRENCHMARK supports sequential, MPI and Ibis jobs as well as some built-in synthetic applications which are parameterisable for computation, communication, memory and I/O requirements. It provides many statistical distributions for modelling the inter-arrival time of jobs, including uniform, normal, and Poisson. It is easy to generate complex and diverse workloads. We have used GRENCHMARK to conduct experiments with KOALA and to test the reliability of its runners. GRENCHMARK has been developed at the Technical University of Delft by A. Iosup.

Figure 2.4 shows an overview of the GRENCHMARK process. A workload can be generated with different application types. GRENCHMARK will then submit and monitor the workload. It can submit to KOALA as well as SGE and Condor. After the jobs have run GRECNHMARK will collect the output and optionally perform some analysis on the results.

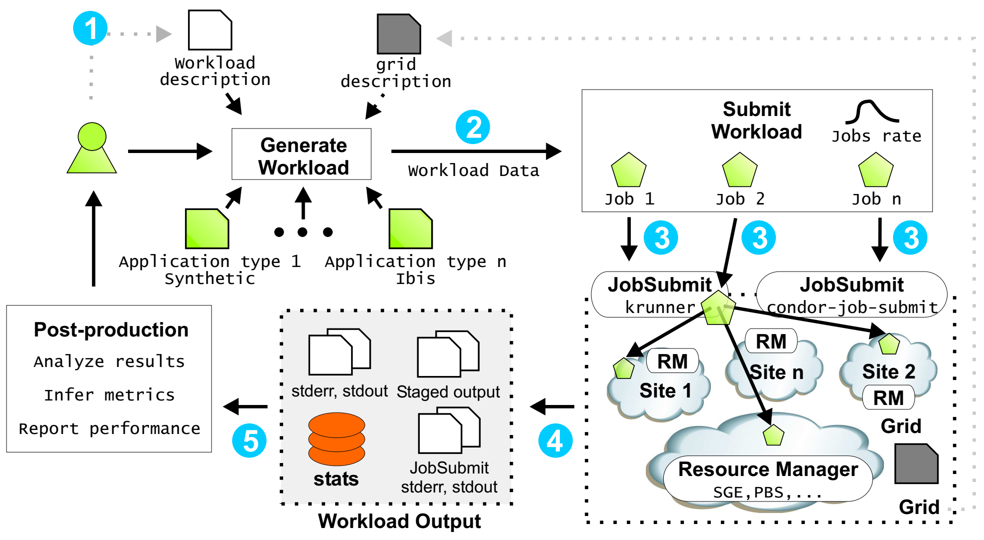


Figure 2.4: An Overview of GRENCHMARK

Chapter 3

The Koala Grid Scheduler

The KOALA co-allocating grid scheduler is developed at the PDS group at Delft University of Technology. As far as the author is aware not many real implementations of a grid schedulers exist. Much research has been done for grid scheduling [10, 41, 27, 45, 6, 15], but of it uses simulations and do not take into account difficulties encountered in real implementations. Some grid schedulers are actually no more than simple submission tools or are load balancers at best. For instance, the Globus Dynamically-Updated Request Online Coallocator (DUROC) [17] is able to submit co-allocated jobs but does not schedule, i.e., the user has to specify execution locations. Scheduling co-allocated jobs is yet another difficulty which has not been solved very often.

KOALA is able to schedule and submit co-allocated jobs as well as other complex job types. It also implements some fault tolerant behaviour to overcome the many errors encountered in a grid environment. Most of the scheduling research for KOALA has been done by H. Mohamed of the PDS group in Delft [36]. This report focuses mainly on the submission and managing part of running jobs with KOALA, although large parts of KOALA have been implemented in cooperation between the author and H. Mohamed.

We will start with a small introduction to KOALA in Section 3.1 followed by discussing the design of KOALA in Section 3.2. Some internal mechanisms of the scheduler are detailed in Sections 3.3 and 3.4, dealing with placement and claiming policies respectively. A strong point of KOALA, fault tolerance, is discussed in Section 3.5. We discuss the information service of KOALA in Section 3.6 followed by the data management component in Section 3.7. In Section 3.8 we start with the main contribution of our own work, the runners. KRunner is discussed in Section 3.8.3 and we end this chapter with some experiments in Section 3.9.

3.1 Introduction

The KOALA co-allocating grid scheduler has been developed to efficiently and easily schedule parallel applications on multiclusters (or grids in general) in which the local schedulers do not support reservation mechanisms [36, 35, 37]. KOALA is written in Java 5.0 and supports total, fixed, semi-fixed, and non-fixed jobs. KOALA is part of the PhD thesis of H. Mohamed.

KOALA separates the job submission procedure into two parts. Job components are first placed and subsequently claimed. Placement here means that a pair of execution site-file site has been determined according to the scheduling policy and that data move operations have been initiated to transfer necessary files. Processors are claimed after all job components have been placed. Between job placement time and job claiming time the processors could be allocated to some other job, if this happens that job component is re-placed on another site. This Incremental Claiming Policy (ICP) avoids wasting CPU cycles which waiting job components would otherwise use. To prevent a job from never starting, i.e., starvation, the time between job placement time and job claiming time is decreased by a fixed amount after every claiming failure.

A job can fail for various reasons, e.g., badly configured or faulty nodes, hardware, and software errors. KOALA resubmits a failed job and counts the number of failures of the supposedly faulty node. When a job fails a previously set number of times (can be ∞) then the job is removed and not rescheduled. If the error count of a node exceeds a fixed number then that node is not considered by the co-allocator anymore.

3.2 Design

KOALA is divided into four parts, the co-allocator, the information service, the data mover and the runners. Users can choose between different runners in order to run their jobs. Currently three runners have been implemented:

- **KRunner**, the KOALA runner (default)
- **DRunner**, a runner using DUROC
- **GRunner**, a runner for Ibis jobs
- **IRunner**, an alternative runner for Ibis jobs

The runner requests placement from the central co-allocator. The co-allocator finds available sites for the job to run using the KOALA grid information service (KGIS). KGIS collects information about the grid from multiple other services, e.g., NWS, MDS [59], etc. Once suitable locations have been found the runner initiates necessary file transfers and other setup procedures. The data mover is used to transfer large input files. It is also capable to do 3rd party transfers, i.e., it can transfer files between two remote sites instead of just being limited to transfers between the local and a remote site. At claiming time the co-allocator notifies the runner again, which will now submit the job to the system using Globus components [4].

Figure 3.1 shows an overview of the components of KOALA. A job is specified using the Resource Specification Language (RSL). Jobs can, amongst others, specify their executable, input and output files, requested number of processors and environment variables using RSL. Submitting a job starts with constructing a Job Description File (JDF) which is usually expressed in RSL. The box marked RunnerShared (RS) depicts the part of KOALA running on the machine of the user. The users chooses one of the above mentioned runners to submit his job. The top box holds all elements of the co-allocator. Iperf, the Monitoring and Discovery Service (MDS), and the Replica Location Service are part of KOALA's information system. With the help of these the

Co-Allocator (CO) finds suitable locations. The Job Dispatcher (JD) starts claiming the components and instructs the Data Mover (DM) to transfer files from the File Sites (FS) to the Execution Sites (ES).

Figure 3.2 depicts a more symbolic overview of KOALA. The co-allocator is a queue-based scheduler.

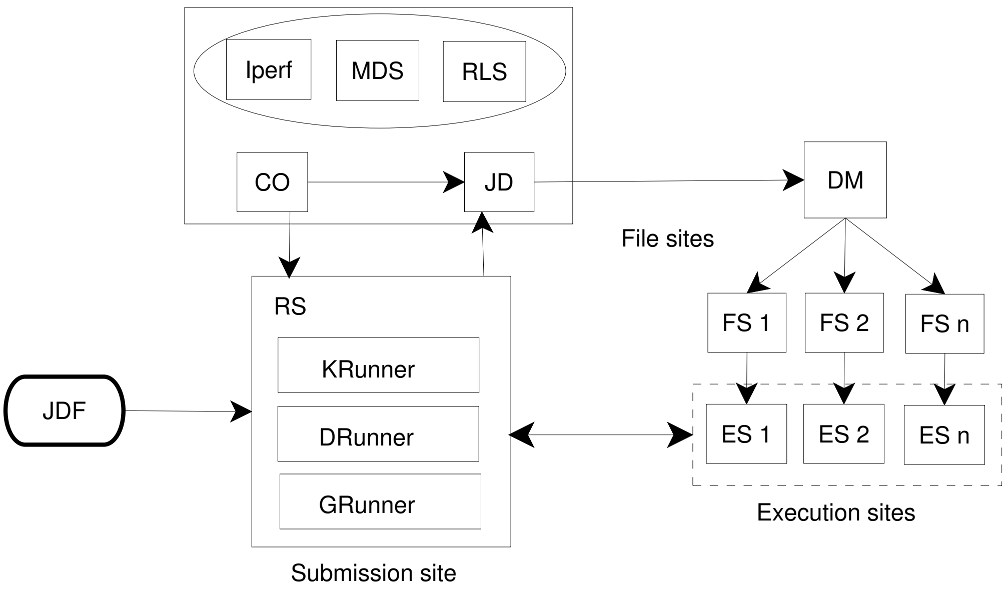


Figure 3.1: An overview of the architecture of KOALA.

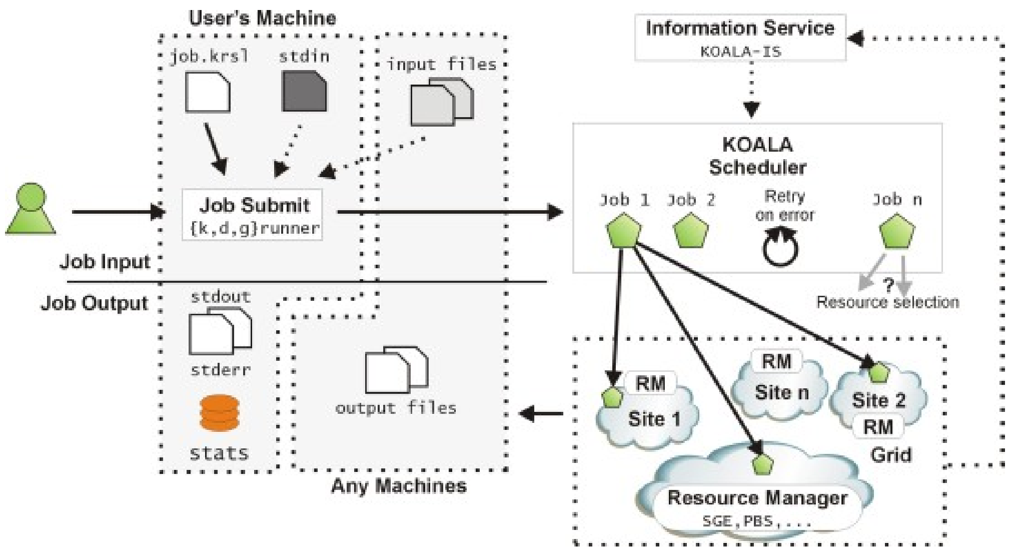


Figure 3.2: A symbolic overview of the job flow in KOALA.

3.3 Placement policies

New jobs are inserted into the low or high priority placement queue. These queues are scanned periodically for jobs that can be placed. Placed jobs then move to the claiming queue and finally to the running queue. Jobs in the placement queues are scanned First In First Out (FIFO). If a job cannot be placed then the next one in the queue will be tried. This mechanism is better known as *backfilling*. If the number of placement tries for a job exceeds a certain limit (which can be ∞) then that job will be removed from KOALA. This selection policy favours a high utilisation of the system instead of minimising job response time.
The two following placement policies have been implemented in KOALA for trying to place a job (in the literature this is referred to as allocation policy):

- The **Worst-Fit (WF)** policy is a well known policy which places a component in the cluster with the largest available free space. The advantage of WF is its automatic load-balancing behaviour. A disadvantage in practise is that large jobs have less chance of placement because WF tends to reduce large free processor spaces.

- The **Close-to-Files (CF)** policy uses information about the presence of input files to decide where to place jobs. Sites with the necessary input files already present are favoured as placement candidates, followed by sites for which transfer of those files take the least amount of time.

When placing a job its job components are sorted in descending order of job component size (size being the number of requested processors).

3.4 Claiming policies

KOALA uses the Incremental Claiming Policy (ICP). Before a job can start, its input files need to be transferred to the execution site. ICP waits with claiming the resources, i.e., the processors, until the files have been transferred. Claiming the resources right after placement means that those resources will be idle while the files are being transferred, which degrades system utilisation. Postponing claiming means that between placement and claiming of the resources there is a period where other jobs can still claim these resources. When resources are not available when the actual claiming is done ICP tries to replace the component which cannot be claimed. At every claiming retry the waiting time between placement and claiming is reduced. ICP tries to minimise wasting resources this way but makes sure the component will be placed in the end. ICP keeps retrying to claim the resources until it succeeds or until a certain limit is reached.

ICP avoids wasting processor time when large files are being used. These files can sometimes be in the order of tens of gigabytes or even terabytes. In the case of co-allocation there is even a bigger chance of processor wastage when one component is scheduled on a cluster with a slow network connection or bandwidth.

Figure 3.3 shows the process of the submission of a job. After a job is submitted (A), KOALA tries to place the job on available clusters (B). Subsequently KOALA tries to claim the resources for each component (C). After all components are claimed the job is started (D). If there are input files that need to be transferred before the Job Start Time (JST) then KOALA wait with starting the job until the estimated File Transfer Time (FTT) has passed. When KOALA fails to claim the resources for a component, additional claiming tries are done, possible on different clusters. The time from when a component is claimed but it cannot be started yet because it has to wait for other components is Processor Wasted Time (PWT). The placement and claiming time together form the Total Waiting Time (TWT). The Job Claiming Time (JCT) is different for every component, KOALA tries to minimise the PWT for every component.

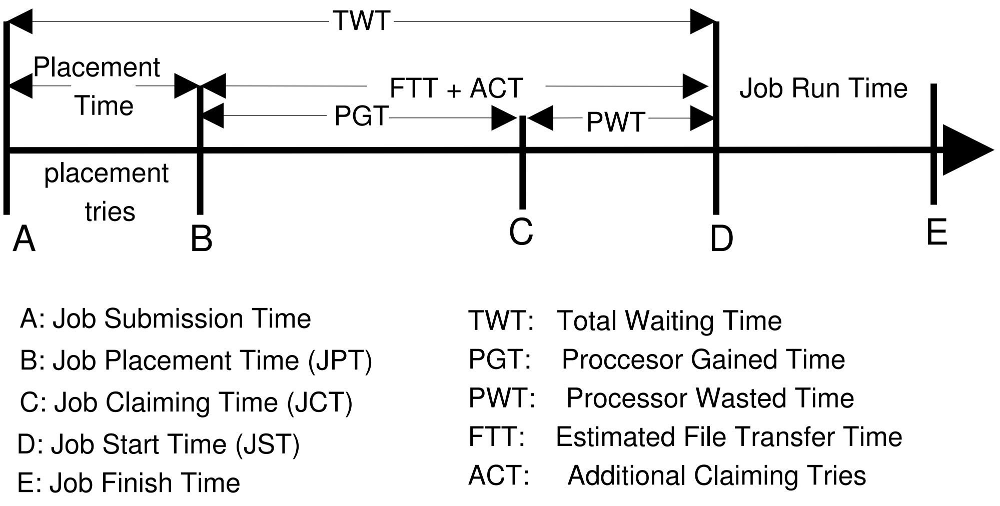

TWT
Placement Time
FTT + ACT
PGT
PWT
Job Run Time
placement tries
A
B
C
D
E
A: Job Submission Time
B: Job Placement Time (JPT)
C: Job Claiming Time (JCT)
D: Job Start Time (JST)
E: Job Finish Time
TWT: Total Waiting Time
PGT: Proccesor Gained Time
PWT: Processor Wasted Time
FTT: Estimated File Transfer Time
ACT: Additional Claiming Tries

Figure 3.3: The time line of a job submission in KOALA.

3.5 Fault tolerance

Grid systems can fail in many different ways. Errors can occur through hardware failures, software errors, configuration mistakes, etc. When an error occurs KOALA records the cluster which failed. KOALA tries to detect which failures are likely to be a cluster failure and which are one-shot errors (a transient glitch). When there is a persistent failure KOALA will not consider the malfunctioning site in the placement procedure anymore.

KOALA also determines if a job can be resubmitted after a failure. The job will be added to the front of the placement queue and will possibly be relocated to another site. Jobs that fail due to errors by the user, e.g., missing input file, will be removed from KOALA.

3.6 KGIS

The Koala Grid Information System (KGIS) provides KOALA with information about the availability of processors in the DAS. Included in the Globus toolkit there is the Monitoring and Discovery Service (MDS), which is a service that provides information about the resources in a grid. MDS is based on a LDAP information service, in our experience MDS was too slow to keep up with the changes in availability of processors in the DAS. During the daytime the maximum execution time of a job is 15 minutes in the DAS, the granularity of updates in MDS was too large to handle this.

To obtain up-to-date information we query the local schedulers directly. Both openPBS and SGE provide an interface to resource availability through a command line tool called qstat. While using openPBS the output of this command was interpreted by a shell script, which was run on demand by the KGIS. SGE's qstat is able to provide detailed XML output. We adapted KGIS to be able to parse and interpret this XML data. The qstat command of SGE has one drawback in that it cannot query remote clusters, to solve this the KGIS sets up a secure shell (ssh) connection to the cluster it wants to query before executing qstat. KGIS queries the clusters on demand, i.e., whenever the scheduler needs information the KGIS starts to query the respective clusters. Combined with the ssh connections this on-demand policy did not work very well, when many jobs are submitted, the KGIS opens too many ssh connections which affects scheduling performance considerably. To solve this the KGIS queries all clusters periodically, the information is stored in a cache, whenever the scheduler queries the KGIS it is serviced from this cache. When there are no jobs in KOALAs queues the KGIS does not update the cache at all. This asynchronous method of providing information helps in decreasing the load on the clusters.

3.7 The Data Mover

Jobs can require, possibly very large, input files before they are able to run and the jobs output files need to be transferred back to the user. KOALA implements two methods of data transfer, the first is Globus Access to Secondary Storage (GASS), the second is GridFTP. KOALA uses GASS to handle the standard process streams, i.e., stdin, stdout, and stderr. It also uses GASS to transfer the executable of the job to the execution site. For larger files it pays of to use the relatively heavy GridFTP service, which is also developed by Globus. We use GridFTP to transfer large input files as well as transfer the output files of the job. GridFTP is optimised for transfer of large files; it can transfer files in a parallel mode, this means it opens multiple connection for one transfer and divides the transfer of data over these connections. It also supports resuming of broken transfers and a way of controlling the bandwidth used during transfer. GridFTP is secured by the Grid Security Infrastructure which makes it a heavyweight process.

3.8 Runners

There are many types of jobs which can run on a grid, e.g., large batch jobs, co-allocated MPI jobs, etc. In order to support different job types KOALA has a *runner* for every different job type. Currently there are the KRunner, the DRunner, and the GRunner. The KRunner is KOALAs default runner. The DRunner uses the Globus component DUROC [29] to submit its jobs. Finally, the GRunner is used to submit Ibis jobs. Section 4.1 will go into more detail about Ibis. The GRunner is described in Chapter 4.

KOALA can easily be extended with more runners. A clean Java interface exists for implementing a KOALA runner. Most runners will not need any extra functionality from the co-allocator. This architecture enables the generic co-allocator to be used by any specific job type. We plan to add support for at least two other runners in the future, the MRunner which can run malleable jobs and the PRunner (name?) which is suitable to run parameter sweep applications.

A KOALA runner is responsible for communicating with the co-allocator and for submitting the job to the system. The communication with the co-allocator is defined in the *submission protocol*. The actual deployment of the job to the system we will call the *submission mechanism*. By default we use the Globus component Grid Resource Allocation and Management (GRAM) to submit jobs to the system.

3.8.1 Submission protocol

A KOALA runner goes through the following steps:

1. Parse job request (using RSL or a runner-specific format)
2. Send a new job request to the co-allocator
3. Receive a KOALA job identifier
4. Await placement, receive execution site
5. Transfer files, if necessary
6. Await submit order from co-allocator
7. Submit the job to GRAM
8. Listen for GRAM status messages
9. Send execution result to co-allocator
10. If errors occurred, update system information
11. Resubmit job, if necessary

These steps are implemented in a basic structure from which all runners inherit. For every step where a runner could have specific needs, e.g., using a runner-specific way for defining a job or a different way of transferring files, a callback exists. A customised runner will be able to use these callbacks to define runner specific behaviour. A runner just has to inherit from the RunnerShared Java object to do this.

Awaiting placement and submit orders from the co-allocator are steps that are performed separately for each component of the job. When a job finishes, either successfully or in error, the co-allocator is notified of this result.

3.8.2 Submission mechanism

The runner itself finally submits the job to the grid after receiving the necessary information from the co-allocator. Currently, we use the Globus Toolkit for submission to the grid, although the separation of runner and co-allocator enables us to use different *grid middleware* by only implementing a new runner. We use the Java Commodity Grid Kit, a Java binding to the Globus components.

Grid Resource Allocation and Management (GRAM) is the Globus component which manages single resources in a grid. GRAM submits, controls, and monitors jobs running at a resource. The submission of a job consists of three phases:

1. Authentication
2. Creation of the Job Manager
3. Setup monitoring and reporting

First, the user has to authenticate herself at the gatekeeper. The gatekeeper validates the credentials of the user and, if successful, creates a Job Manager. The Job Manager then requests the necessary resources at the local resource manager (i.e., usually this resource is a processor) and starts the job. The Job Manager starts monitoring the progress of the job. A GRAM callbackhandler can be registered with the Job Manager in order to receive status updates. It is possible to register more than one callbackhandler. One is automatically created when a GRAM job is submitted, but more of them can also be created explicitly. This process is illustrated in Figure 3.4.

After a job is submitted to it, GRAM will return a job identifier with which we can monitor and control our job. The callback handlers are installed in the user program and are used to keep the user program informed of the state of the job. Possible (GRAM) job states are:

- Unsubmitted
- Stage in
- Pending
- Active
- Done
- Stage out
- Failed
- Suspended

The input and output of job can be retrieved using the Globus Access to Secondary Storage (GASS) server. The runner starts a GASS server and passes its URL along when submitting a job to GRAM. All the input and output of the remote job will now be redirected to this URL and consequently the GASS server. The GASS component can be used to transfer files as well as the standard input and output (which in *nix terms are both file descriptors anyway). Figure 3.5 shows a simple scenario using the default runner.

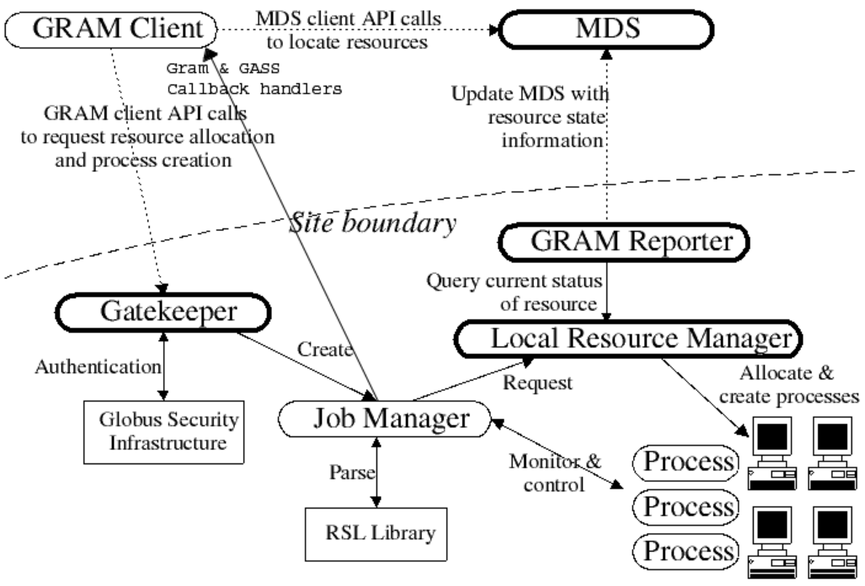

GRAM Client
MDS client API calls to locate resources
MDS
Gram & GASS Callback handlers
Update MDS with resource state information
GRAM client API calls to request resource allocation and process creation
Site boundary
GRAM Reporter
Query current status of resource
Local Resource Manager
Gatekeeper
Create
Request
Allocate & create processes
Authentication
Globus Security Infrastructure
Job Manager
Parse
Monitor & control
Process
Process
Process
RSL Library

Figure 3.4: Major components of the GRAM implementation. Those represented by thick-lined ovals are long-lived processes, while the thin-lined ovals are short-lived processes created in response to a request. *(Taken from [16], but slightly edited.)*

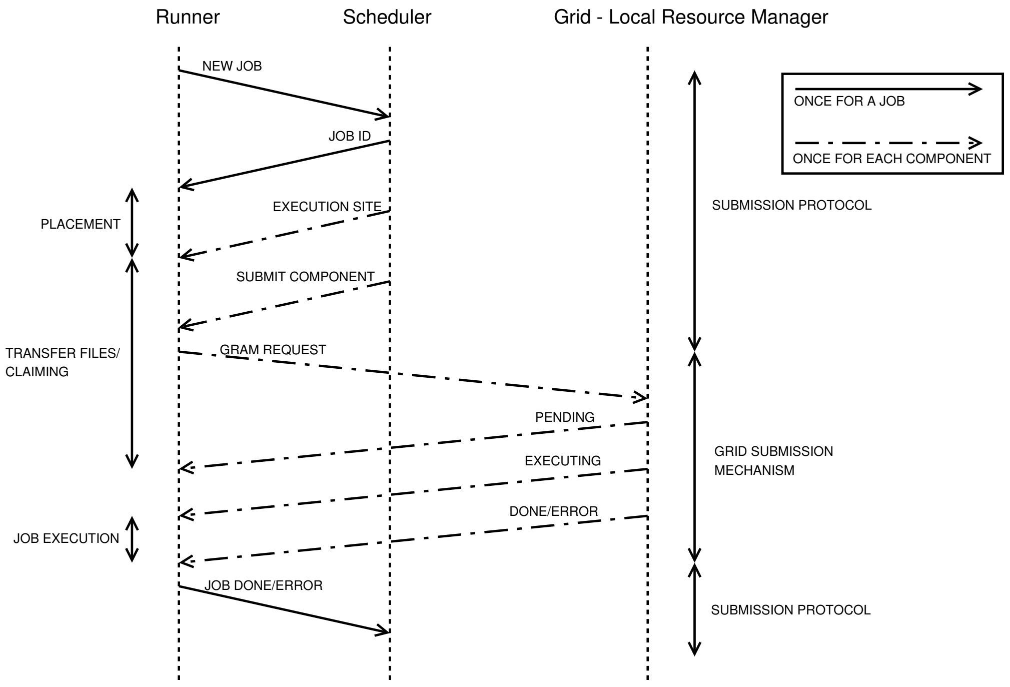


Figure 3.5: Possible scenario for the KRunner.

3.8.3 The KRunner

The KRunner is the default runner for KOALA, it is capable of running jobs that do not require special startup procedures. When running co-allocated jobs with the KRunner it means that each component will be executed independently, it is up to the application to handle the communication and startup barrier. The KRunner does support file staging, it transfers input files before the job executes and retrieves the output files when the job is done. Although it does not implement a sandbox, it is possible to create a limited sandbox using the file transfer capabilities. The KRunner is meant to be a bare bones runner which gives the user much room to customise and control the run.

The KRunner is also used as a basis for other runners, it gives the basic implementation a runner needs to be able to interface with the KOALA co-allocator.

3.9 Experiments

We have run many tests to ensure the reliability of the KRunner. We used GRENCHMARK to generate workloads containing many variations of jobs, i.e., fixed and non-fixed jobs, many components or just a single component, and various sizes for these components. As a test application we use a simple parallel MPI program which calculates pi. In this section we discuss four of the experiments that we have run with the KRunner.

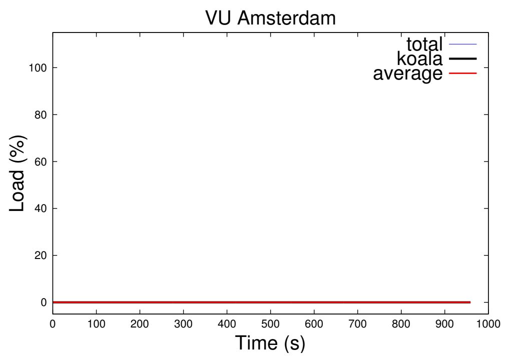
3.9.1 Experiment 1

In the first experiment we submit 50 fixed jobs. All jobs consist of a single component and the size of each component ranges from 2 to 20 processors. GRENCHMARK randomly assigns sizes and execution locations to job components. The inter arrival time of the jobs is based on the Poisson distribution with a mean of one second. All jobs were started from the same submission site. In Figure 3.6 the results of experiment 1 are shown.

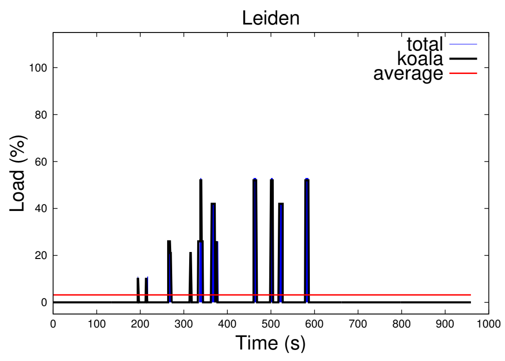
The filled surface displays the total load of a cluster, with exception of Figure 3.6(f) which shows the average load of all clusters combined. The black line depicts the load our jobs added to the DAS. The horizontal line is the average load during the whole experiment. All jobs completed successfully. The Amsterdam (VU) cluster was not used during the experiment due to a persistent failure and was removed from the selection process of the co-allocator. We observe that the Delft cluster was the only cluster with some background load. The average job completion time was less than one minute. We also observe that due to randomly assigned execution locations by GRENCHMARK we see that the jobs are fairly distributed over all clusters.

3.9.2 Experiment 2

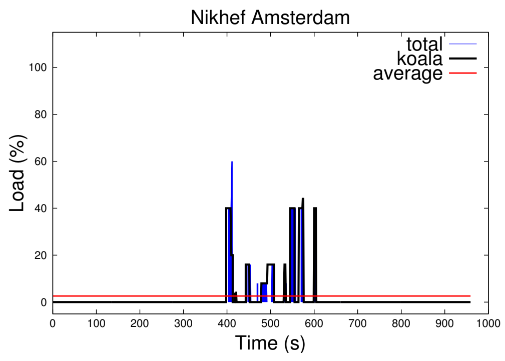
In the second experiment we run 100 non-fixed jobs, with the same application as used in Experiment 1. All jobs consist of a single component with size varying from 2 to 20 processors. We observe that there was a little more background load during this experiment but that it is not hindering our results. The main result of this experiment is that even though the jobs are still distributed over all clusters, the distribution is not as



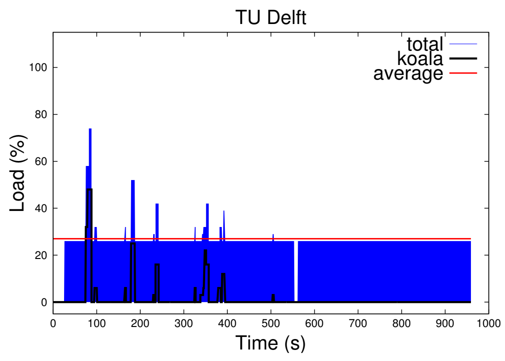
(a)



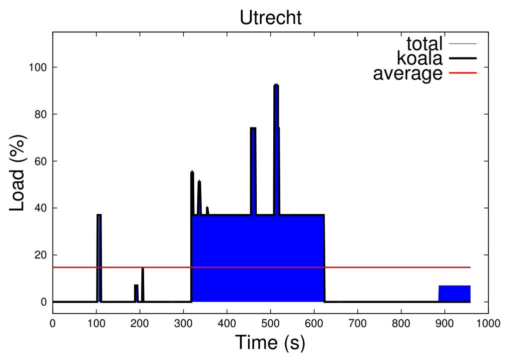
(b)



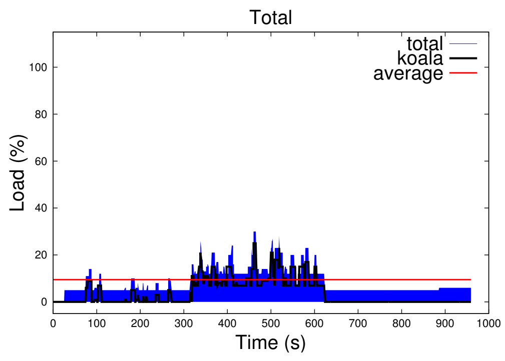
(c)



(d)



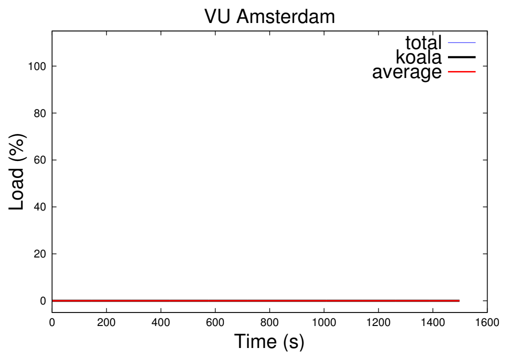
(e)



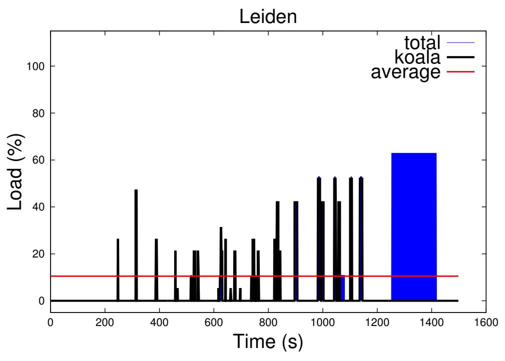
(f)

Figure 3.6: The system load of Experiment 1.

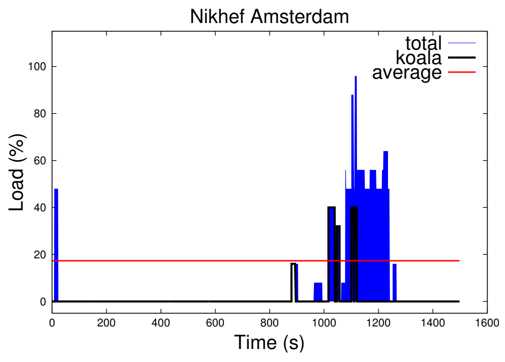
fair as in Experiment 1. The reason for this is that KOALA has now scheduled the jobs instead of GRENCHMARK assigning random locations to the jobs. KOALA's scheduling policy favoured the Leiden cluster during the time of this experiment, the reason for this is the Worst-fit scheduling policy which places jobs in the cluster where the most processors are available. The Leiden cluster had the most processors during this experiment and therefore it received most of the jobs. The results of this experiment are shown in Figure 3.7.



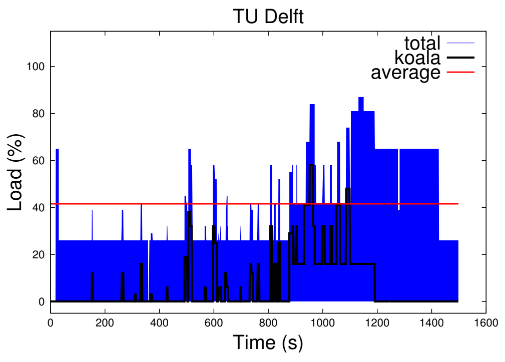
(a)



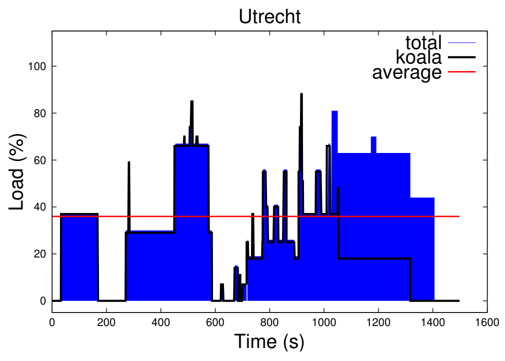
(b)



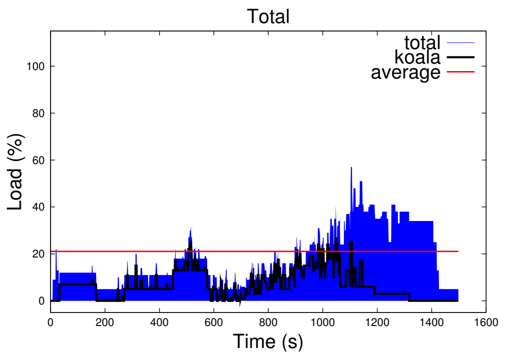
(c)



(d)



(e)



(f)

Figure 3.7: The system load of Experiment 2.

3.9.3 Experiment 3

In the third experiment we have run a mix of 50 fixed and 50 non-fixed jobs. The size of the components again range from 2 to 20 and the number of components per job ranges from 1 to 4. All jobs completed successfully except one, which failed due to Myrinet network problems. We observe a considerable background load in the Leiden and Delft cluster due to long running jobs of other users. KOALA is able to fairly schedule this mix of fixed and non-fixed jobs. The results of this experiment are shown in Figure 3.8.

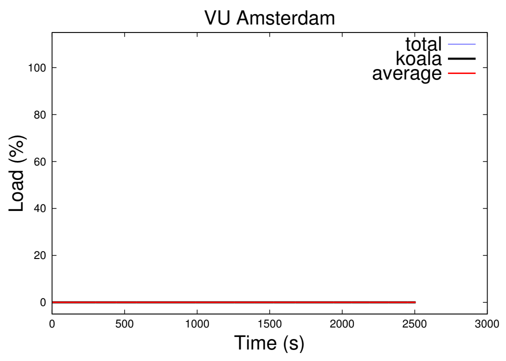
3.9.4 Experiment 4

In the last experiment we tested the fault tolerance of KOALA. We submitted 100 non-fixed jobs with the same configuration as the previous experiment. We introduced an artificial error in all clusters, except the Amsterdam (Nikhef) cluster. KOALA first tries to schedule the jobs evenly among all clusters. When an (artificial) error occurs the error count of the cluster is updated until it reaches the threshold level and is not considered by the co-allocator anymore. All clusters, with the exception of the Amsterdam cluster, reached the threshold level, after which all jobs completed successfully in the Amsterdam cluster. This illustrates the capability of KOALA to 'get the job done' even in error prone environments.

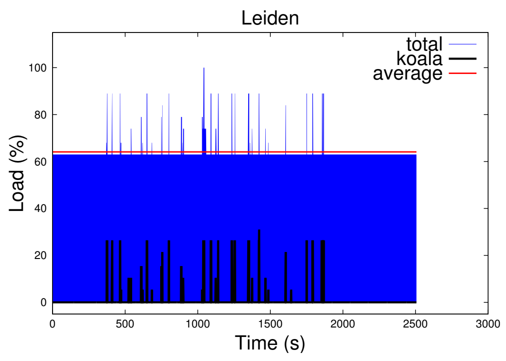
3.9.5 Additional experiences

In addition to these experiments, we have gained much more experience with running jobs with KOALA from other projects of master students. For instance, we have run over 200 jobs for a parallel implementation of the chess game and have run a molecular dynamics application called Chromax for another master's project.

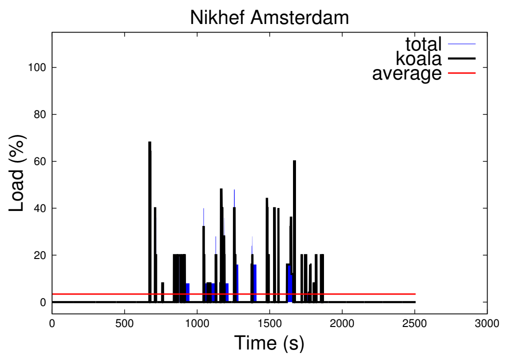
While stress testing KOALA with very large batches of jobs we experienced three difficulties with KOALA, and more generally with grids. The first is that running many jobs with KOALA means running many Java virtual machines (JVM), because each runner requires its own JVM. Running over 200 JVMs can be very demanding on the submission machine. The second difficulty while running many jobs from the same submission machine (file server) is the number of network connections which are opened. A single runner can open many network connections due to communicating with the co-allocator, communicating with GRAM, and performing file transfers. The last difficulty becomes apparent when the whole DAS is heavily loaded and most processors are occupied. In this case jobs start being queued at the local resource managers, since KOALA is always searching for idle resources it will never submit jobs to the queue of a local resource manager. In a large enough grid there is a high probability of finding idle resources but this behaviour makes KOALA 'too nice' in smaller, heavily loaded grids.



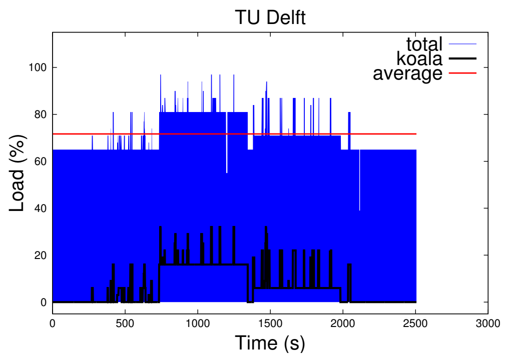
(a)



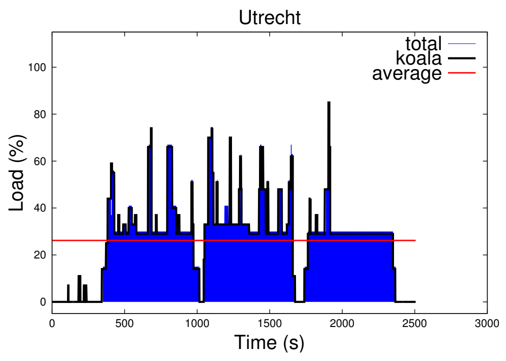
(b)



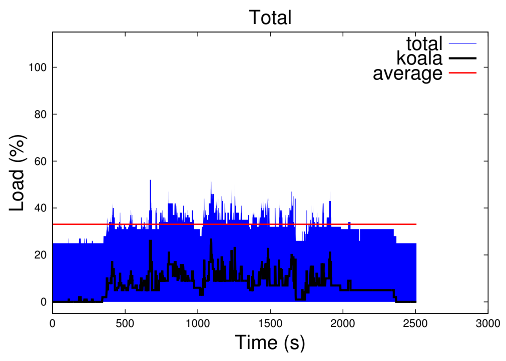
(c)



(d)



(e)



(f)

Figure 3.8: The system load of Experiment 3.

Chapter 4

Support for Ibis jobs in Koala

The biggest bottleneck in parallel programs tends to be communication. Maximising performance often means minimising the time spent in communication of the application. Of course, communication is not the bottleneck in CPU-bound or embarrassingly parallel applications. In a multicluster system, and even more in a grid, communication is an even greater bottleneck due to the relatively slow communication channels between remote sites. A grid scheduler could avoid using these slower channels by assigning all components of a job to a single cluster or execution site, but this would counteract the increase in utilisation which we are trying to accomplish with co-allocation. Moreover, co-allocation enables us to run jobs that need more resources than a single cluster can accommodate. Ibis is a Java communication library which is optimised for wide area network connections. For this reason, we believe that support for the Ibis wide-area communication library in KOALA is very useful and we have implemented a runner which can submit Ibis jobs. Implementing the runner for Ibis also led us to test and refine the architecture of KOALA. We implemented the GRunner because communication is an important factor in co-allocated jobs and to show that KOALA can handle many different kinds of jobs.

This chapter will describe the design and implementation of the GRunner. We will start with discussing Ibis itself in Section 4.1. The Ibis distribution comes with its own submission tool called *grun*, which has been developed at the Vrije Universiteit Amsterdam. We will be using grun as the submission mechanism and will describe grun in Section 4.2. In Section 4.3 we will state the objectives of the GRunner, followed by a few considered, but rejected, design approaches in Section 4.4. The design of the GRunner is given in Section 4.5. Some implementation details are discussed in 4.6. We conclude with the experiments we have performed with the GRunner in 4.7.

4.1 Ibis

The Ibis project is designed to provide an efficient and flexible Java-based programming environment for grid computing. Java's support for distributed computing, Remote Method Invocation (RMI), has important shortcomings for high-performance grid computing: it is difficult to implement efficiently and only expresses client-server style communication [52]. Ibis overcomes these shortcomings on client-server style communication, as well as provides other communication paradigms, like group communication.

As a default, Ibis uses communication over TCP/IP. Ibis improves on RMI by avoiding the high overhead of runtime type inspection using several optimisations. Ibis does not use native code to accomplish this and thereby adheres to the Java portability philosophy. It is possible though to load a more efficient communication implementation, whenever available, which is usually written in native code. The TCP/IP implementation will run ”everywhere”, but, for example, on the DAS-2 with its Myrinet network, the considerably faster Glenn's Messages (GM) proprietary communication layer can be used. [52].

The top layer of the Ibis system consists of the Ibis Portability Layer (IPL). The IPL consists of a set of Java interfaces that defines how an Ibis application can make use of the Ibis components. The Ibis application does not need to know which specific Ibis implementations are available. It just specifies some properties that it requires and the Ibis system selects the best available Ibis implementation that meets these requirements [52].

All communication connections in Ibis are conceptually unidirectional. For two-way communication, two connections have to be established. The IPL provides Send and Receive Ports for this. Although the connections are unidirectional, they can be many-to-one or one-to-many connections, i.e., one Send Port can be connected to multiple Receive Ports and vice versa. An Ibis application registers its Ports at a central Ibis nameserver. Connecting Ports have to share the same properties as registered at the nameserver.

4.1.1 The Ibis programming models

Next to the classic RMI-style programming model, a few other programming models have been implemented on top of Ibis:

- **Group Method Invocation (GMI)** is a generalisation of the RMI model. It allows methods to be invoked on single objects or a group of objects. Result values can be combined into single results. The different method invocation and reply handling schemes can be combined orthogonally, allowing us to express a large spectrum of communication mechanisms of which RMI and MPI-style collective communication are special cases [2].

- **Replicated Method Invocation (RepMI)** offers object replication aimed at improving the performance of parallel Java programs.

- **Satin** is an extension of Ibis with Cilk like primitives. Satin has been developed to easily program divide-and-conquer style algorithms on wide-area systems. The Satin runtime system hides these wide-area optimisations from the user program [55, 56].

These programming models plus Java's ‘run everywhere’-principle make Ibis a good alternative for parallel and grid programming. For these reasons we want KOALA to support Ibis jobs.

4.1.2 Name- and poolserver

A nameserver is needed to run Ibis jobs. This is a central component which coordinates the setup of communication Ports between the Ibis instances. The poolserver provides MPI-like ranking functionality. The Ibis runtime system is capable of detecting when a job component crashes or leaves the run as well as letting new Ibis instances join the run. This enables Ibis programs to implement fault tolerance schemes at the application level.

4.2 Grun

Grun is a submission tool designed to simplify the submission of Ibis jobs to a grid. A limited form of fault tolerance is supported by a simple retry feature when job submission fails. Grun only supports fixed jobs.

Grun is packaged with the Ibis distribution and is the first submission tool to support Ibis applications. Grun is written in Python and uses the Python interface to Globus components called pyGlobus. It handles staging through the GASS server and submits jobs using GRAM. When invoked with the correct arguments, grun will start and stop the Ibis nameserver as needed. A sandbox is created within the users home directory in which the job will run, this ensures no data will involuntarily be overwritten.

In order to support Ibis-jobs with KOALA, we designed a wrapper around the grun submission tool. Although a direct implementation for support of Ibis is possible, wrapping grun will sustain Ibis support in KOALA with new versions of Ibis without the need of re-implementing Ibis support at every update. A direct implementation without grun may still see the light of day though, since the KOALA framework is easily extended with different submission modules, i.e., runners.

The IRunner is an alternative to the GRunner for running Ibis jobs. It is based on the KRunner, the IRunner ‘manually’ manages the startup of the nameserver and starts the Ibis job through an extensive RSL job description file. Configuring this job file requires much knowledge about running Ibis jobs and may thus not be suitable for most users, it does provide much control for the user though if he wishes that.

4.3 Objective

Wrapping grun will hide some difficulties for the submission of Ibis jobs but will also introduce some new problems. We need to obtain the input and output from the user job. Because grun will actually submit the job, our runner will not be informed of job states and thus cannot monitor and control the job anymore. Another deficiency is that grun only supports fixed jobs.

Our *objective* therefore is to construct the GRunner which is able to:

- redirect job I/O
- trap job state messages
- submit non-fixed Ibis jobs

- use KOALAs fault tolerance mechanisms

Other related problems are how to interface with Python from Java, how to map the job component numbers KOALA uses on grun's own component naming scheme, how to make a grun job request compatible to KOALAs job format (which is RSL) and how to implement all this without the need to patch grun.

4.4 Possible design approaches

Various designs have been explored to construct a mechanism to control grun from KOALA. We will discuss some of these approaches here. We will also discuss why these approaches have been rejected. The main problem is to make our Java program 'communicate' with grun, which is an interpreted Python program. We discuss three explored approaches to this problem:

- **Jython** Jython is an implementation of the object-oriented language Python and is written in Pure Java. It allows us to run Python programs from within a Java virtual machine. With Jython we could run grun inside KOALA and control the Python interpreter step-by-step. Grun uses pyGlobus to interface with Globus components. Jython unfortunately does not support the pyGlobus package. We were told by Jython's creators that no effort will be made to support pyGlobus in the future.

- **CPython & JNI** CPython is a standard mechanism to extend Python with C code. A *shared object*, written in C, can be linked with the Python interpreter. A shared object is the *nix equivalent of the windows *dll*, Dynamic Link Library. Java Native Interface (JNI) is a similar mechanism for the Java language. Sharing code in this way between Java and Python did not work because the CPython shared object could not be combined with a JNI shared object.

- **Screen scraping** Screen scraping is often used to provide new user interfaces for legacy systems. The output of the legacy system is a text-based screen which is continually scanned to extract information. A new user interface then displays the obtained information and also translates user input back to the commands of the legacy system. This scanning of the screen, usually hidden from the user, is called *screen scraping*.

  Starting grun in the background and scanning its output using regular expressions is a way to interface KOALA with grun. We considered we would use this approach only if all others fail. Drawbacks of this approach are its slow performance and a high sensitivity to changes in grun. A lot of maintenance would be required to adapt to changes in the output of grun. Moreover, from a software engineering perspective screen scraping is not a very elegant solution.

4.5 The design of the GRunner

This section will discuss the design of the GRunner. We have implemented a wrapper around grun. We will simply call this the Wrapper. The Wrapper is written in Python and monitors and controls grun. KOALA interfaces with grun through the Wrapper. From KOALAs perspective nothing changes, i.e., the same submission protocol is used. The difference with the default runner is the submission mechanism. Grun will perform the actual job submission through GRAM. The Wrapper retrieves information from grun without the need to explicitly adapt grun to KOALA.

4.5.1 Structure

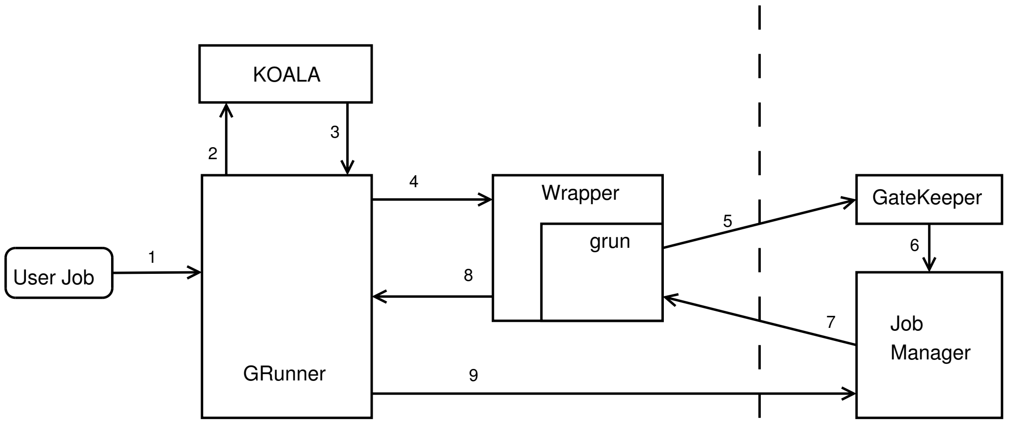


Figure 4.1: The architecture of the GRunner.

Figure 4.1 shows the general structure of the GRunner. The GRunner itself is responsible for translating a grun job request into a KOALA job request. It will construct a RSL specification based on the grun command line parameters (1). The RSL specification is sent to the co-allocator which will find execution sites for the components of the Ibis job (2,3). Once all components are placed the GRunner will start the Wrapper (4). This is an external program in the form of a Python script. The GRunner translates the RSL specification back into grun command line parameters. These modified parameters are fed to the Wrapper. The Wrapper uses grun to submit the job to the system (5). The user is first authenticated at the Globus gatekeeper (6). The gatekeeper will start a job manager for every job (7). The job manager monitors and controls the job at the remote location. The job manager will report successful submission to the system back to grun (7).

Grun will be informed about job state changes by the job manager through a callback mechanism. Instead of relaying these job messages from grun to the GRunner, we wish to receive these state messages directly. GRAM supports this by being able to register more callback handlers at the same job manager. The Wrapper will retrieve the GRAM job component identifier from grun and send it to the GRunner (8). The GRunner then registers itself at the job manager (9) using the job component identifier. The GRunner will be directly informed about any job state changes and the GRunner can also control execution of the job, i.e., decide to terminate it.

The dotted line denotes the boundary between the submission site and the execution sites. On the left side of the dotted line are the components which are running in user space, each instantiated once for every job. The components on the right side are

Globus components running on the remote locations. The components on this side are instantiated once for every component of the job.

Because grun is used to submit the job, we cannot use KOALAs incremental claiming policy. The GRunner waits until all components are ready to claim before it sends the job request to grun. Grun then atomically claims all components at once.

4.5.2 Data exchange between components

This section will go into more detail about communication between the components. An example of command line parameters used to run an Ibis job with grun:

```
grun -v -o
-s "$grun_etc/runsvrs.ibis -name $port1 -pool $port2"
-c "@GRUN_GASS@/$grun_etc/runit.ibis build Main 2000 2000"
-E IBIS_NAMESVR_PORT=$port1
-E IBIS_POOLSVR_PORT=$port2
-p -E GRUN_INFILES="ibis-tree.jar:build-tree.jar"
-P -E GRUN_CLEANUP=1
-E IBIS_ROOT="ibis" -H 2 -m fs0 -m fs1
```

This command line represents a fully-staged Ibis run. Which means that necessary jar files are staged in before execution (ibis-tree.jar and build-tree.jar in this case). The Ibis nameserver is launched externally before the job run. Finally there is a clean up phase after execution in which temporary files are deleted.

The parameters `-N 2 -m fs0 -m fs1` define the job components. In this instance every component requests two processors, one of which is to be run on `fs0` and one on `fs1` (`fs0` and `fs1` are cluster names of the DAS-2). The GRunner will translate this request into a RSL specification, which will look approximately like this:

```
+
(
  &( directory = "/directory/dummy" )
   ( executable = "dummy" )
   ( arguments = "dummy1 dummy2" )
   ( resourceManager = "fs0.das2.cs.vu.nl/sge )
   ( maxWallTime = "15" )
   ( label = "subjob 0" )
   ( maxCpuTime = "15" )
   ( count = "2" )
)
(
  &( directory = "/directory/dummy" )
   ( executable = "dummy" )
   ( arguments = "dummy1 dummy2" )
   ( resourceManager = "fs1.das2.liacs.nl/sge )
   ( maxWallTime = "15" )
   ( label = "subjob 1" )
   ( maxCpuTime = "15" )
   ( count = "2" )
)
```

For non-fixed jobs we extended the grun parameter `-m` to support the special cluster name 'anywhere'. When this cluster name is used, KOALA will treat the component as non-fixed and search for a suitable execution site.

The RSL specification is sent to the co-allocator. Once the job is placed, the RSL will be returned to the GRunner. If 'anywhere' was specified for a component, it is now replaced with the cluster name the component is placed on. In addition to the user provided grun parameters, the GRunner also sends the hostname and port on which a server is listening to receive the job component identifiers from the Wrapper. This server is started by the GRunner before a job is submitted to grun and is called the Wrapper listener. The Wrapper listener waits until it receives the job component identifiers of all the components or until it receives a message indicating the submission has failed.

Once the listener has received a job component identifier, the GRunner registers itself with the job manager. Through a callback mechanism the GRunner will be informed of the job state. The job can now be considered a normal job like with the default runner (KRunner). We can choose to kill the job whenever we want, for instance, when one of the components fails.

I/O redirection

The Globus Access to Secondary Storage (GASS) component of the Globus Toolkit is used to handle all I/O of a remote job. A user has to start a GASS server and pass the URL on which it listens along with the GRAM job request. The RSL specification can now include references to this server when specifying file paths. This can be used to redirect the standard output of all the jobs to a single terminal, or to write output files to the user specified central location instead of on the remote locations. A possible RSL specification could be:

```
  ( stdout = "https://fs3.das2.ewi.tudelft.nl:47378/dev/stdout" )
```

This would redirect the standard output of remote jobs to a terminal on fs3.

For interactive jobs KOALA starts a GASS server and rewrites the RSL automatically. Grun also starts a GASS server to redirect input and output. The Wrapper makes sure all output will be redirected to the GASS server of the GRunner by prepending `stdout` and `stderr` clauses to the RSL that grun submits to GRAM.

4.5.3 Wrapper

When an Ibis job is submitted to KOALA, the Wrapper is started as a separate process. The Wrapper imports grun and starts executing the grun main routine. When importing a script in Python, all variables and functions of the imported script are visible and can be modified. We use this to override key functions in grun with our own functions. After one of our own functions has executed we return control to the respective function in grun. With this approach no changes have to be made to grun but we are able to receive all necessary information from grun through our Wrapper.

For instance, the function `job_submitted_upcall` is defined in grun and is called whenever a job is successfully submitted. The following, very simple, Python code is used to trap this call and execute a function of the Wrapper. The Wrapper returns control to the original grun function when finished and grun will continue with its normal execution. A line starting with the # symbol is comment.

```
import grun

# a Python function in the Wrapper
def redefined_submitted_upcall(job, jobcontact, state, newerr, newerr2 = None):
    # do Wrapping functions here
    .
    .
    # call original grun function
    original_submitted_upcall( job, jobcontact, state, newerr, newerr2 )

# execution of the Wrapper starts here
# save original function
original_submitted_upcall = grun.job_submitted_upcall
# override grun function with our own
grun.job_submitted_upcall = redefined_submitted_upcall
```

In general the Wrapper performs the following actions:

- Read grun Wrapper arguments (KOALA GASS URL + Wrapper listener URL)
- Read (modified) grun arguments
- Store original reference to grun functions
- Override grun functions
- Start grun main routine
- Modify RSL constructed by grun to include KOALA GASS server
- Wait for GRAM job identifier, also called Resource Manager Contact (RMC)
- Report RMC back to Wrapper listener

4.6 Implementation details

Figure 4.2 shows a state diagram of an Ibis job. The states at the bottom depict the *happy flow*, i.e., the states a job goes through if nothing fails. Different errors occur at various states of a job. Depending on the kind of error, KOALA chooses to end the job altogether or to retry the job. Retrying for the GRunner means that an abort command is sent to each component. The GRunner then waits until the Wrapper (and thus the grun) process exits before resubmitting the job to the placement queue.

This might seem like unnecessary overhead, why not resubmit the job immediately without waiting? This is done because grun also starts an external Ibis nameserver which will run on a user-defined port. If we resubmit a job too quickly the new nameserver could fail because it cannot claim its socket. We also wait for grun to finish so it can properly execute its clean up phase in which it removes the temporarily created sandbox.

When a job request with an incomplete or incorrect RSL specification is submitted the job will, naturally, not be resubmitted and will exit immediately. Once all components are placed the claiming phase starts. In contrast to other runners this is done once for the whole job, i.e., the components do not get claimed independently. The claiming is done as a GRAM submission request and can fail for many different reasons, e.g., misspelled or non-existent executable name, input files not present, local resource manager unavailable, etc.

Some of these errors could be caused by the system itself and could be a local phenomenon. In this case the job can be retried. If the error was a temporary failure,

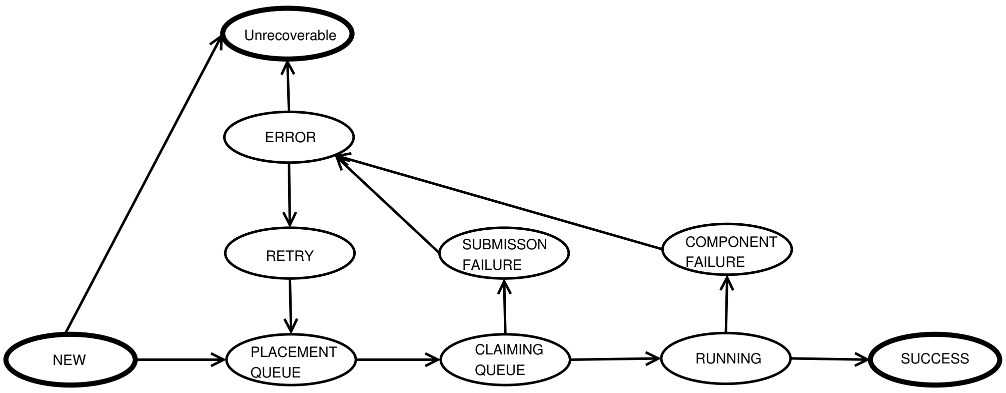


Figure 4.2: State diagram of an Ibis job.

the job could be resubmitted at the same location. If the error persists then the job will eventually be replaced on another site. This could be termed *migration* but there is a subtle difference. The jobs are not checkpointed and restarted, they are resubmitted and started from the beginning. Although this mechanism implicitly load balances the system, no explicit load balancing through migration is done.

If the user himself was responsible for the failure then the job will not be resubmitted. Determining which failures are recoverable and which are not is not a trivial task.

During the runtime of the job any component can experience failure. When one component fails all other components are aborted, even if they were running without problems. Once all components have terminated and the job is removed from the system the GRunner determines whether the failure facilitates a retry or not.

Figure 4.3 shows a scenario of an Ibis job run. It can be compared to Figure 3.5 in Chapter 3 where a scenario of a default job run is given. We can clearly see the overhead of wrapping grun. An overview of the lifetimes of the components involved in the submission process is shown in the sequence diagram in Figure 4.4.

4.6.1 Difficulties with the GRunner

We have tested the GRunner with most of the Ibis applications in the Ibis distribution. The GRunner is of course bounded by the capabilities of grun. Experiments show that all applications that can be run with grun, can be run by the GRunner.

Two rather critical bugs were discovered during testing of KOALAs runners which are worth mentioning here.

GASS server

KOALA interfaces with the Globus components using the Java Commodity Grid kit (CoG kit). The CoG kit is provided by the creators of the Globus Tool Kit and provides implementations of all Globus components in Java. We use the Java implementation of the GASS server to redirect all input and output of a remote job. After many experiments we concluded that this Java implementation contains a critical bug. The GASS server sometimes stops responding after approximately 45 minutes and causes

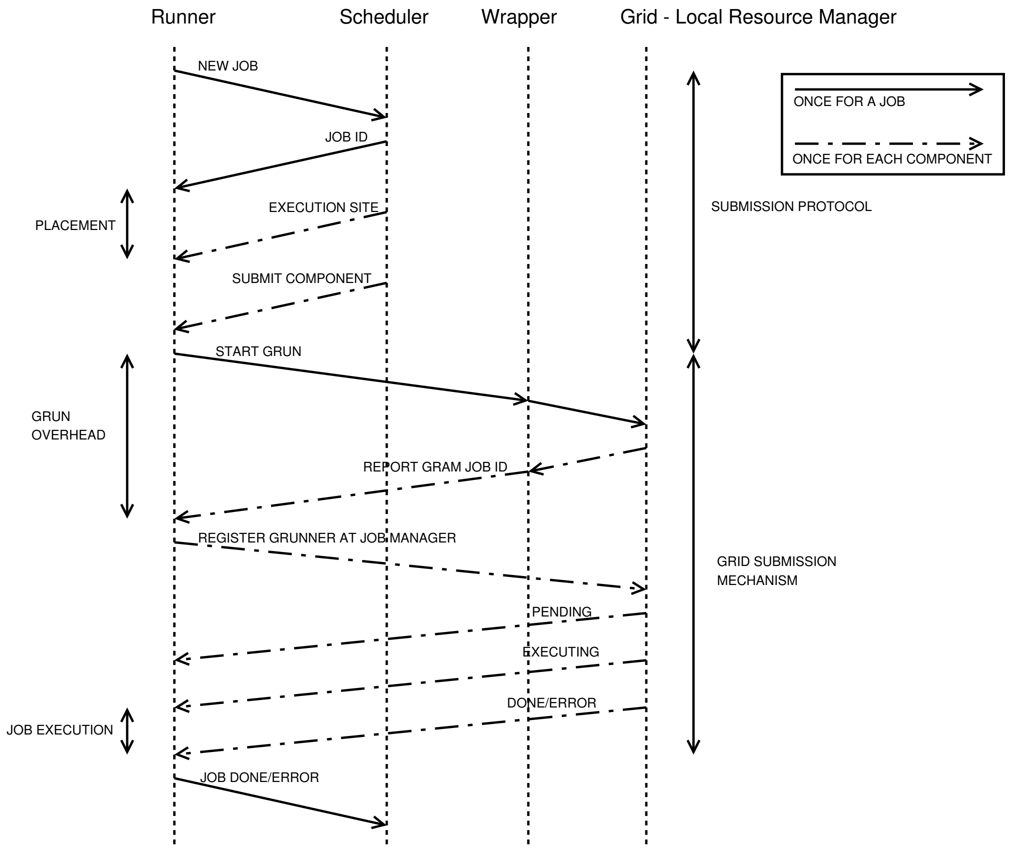


Figure 4.3: Possible scenario for the GRunner.

the GRAM jobs to fail with a `COULD_NOT_STAGE_OUT_A_FILE` error. This fault can be remedied by starting our own GASS server externally instead of through the CoG kit. The command line C-based implementation does not exhibit this error.

Ibis nameserver

Another elusive bug seemed to be the fact that the Ibis nameserver ceases to work when more than approximately twenty processors were requested for one job. Launching the nameserver externally worked like a charm though. It seems this crash only occurs when grun launches the nameserver. When running Ibis jobs 'manually' the name-server performed flawlessly. We can solve this by starting the nameserver directly from the GRunner instead of letting grun handle it.

Note: we have not tested if this behaviour still occurs with the new Ibis distribution (version 1.2).

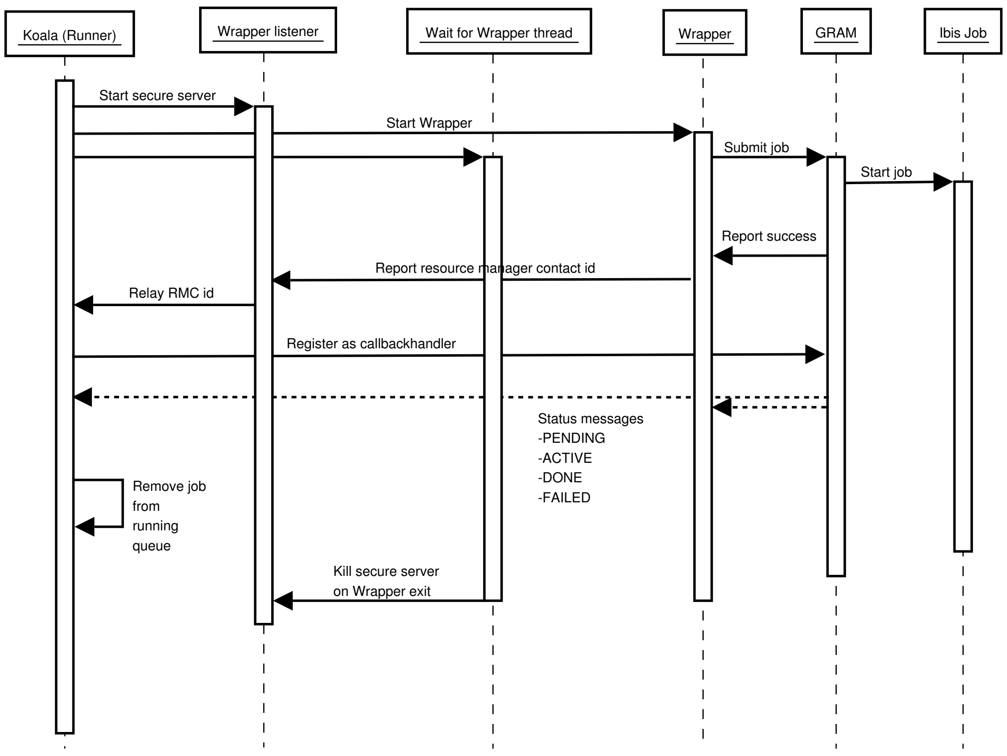


Figure 4.4: Sequence diagram of a successful job submission.

4.7 Experiments

We have run many tests to ensure the reliability of the GRunner and the IRunner. We used GRENCHMARK to generate workloads containing many variations of jobs, i.e., fixed and non-fixed jobs, Ibis and MPI applications, and mixed and Ibis-only workloads. In this section we discuss two of the experiments that we have run with the GRunner and the IRunner.

4.7.1 Test application

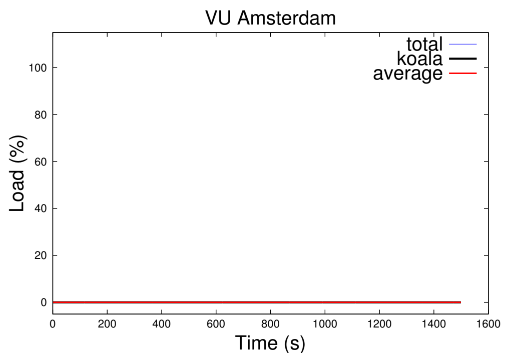
The Ibis application we use in the experiments is the nqueens application. The nQueens problem is to place n queens on a nxn chessboard in such a way that no queens are able to capture each other. We start 10 jobs consisting of 2 components and 10 jobs consisting of 4 components. The 2-component jobs each request 10 processors, the 4-component jobs each request 2 processors, which makes for a total of 20 and 8 processors respectively per job. The average runtime of the application is approximately 2 minutes. All job requests, which were non-fixed, were started from the same submission site without delays between job submissions.

4.7.2 GRunner experiment

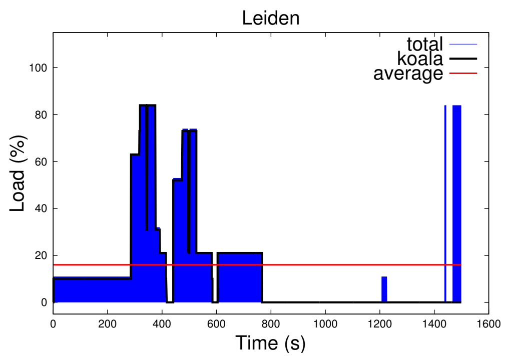
The results of the GRunner experiment are shown in Figure 4.5. First we observe that there was no load on the Amsterdam (VU) cluster, the cause of this is a persistent failure in that cluster which prevents KOALA from scheduling any jobs on that location. We observe that the experiment finished earlier on the Leiden and Delft cluster than on the Amsterdam (Nikhef) and Utrecht cluster. This illustrates the fault tolerance of the GRunner; some jobs failed at first and succeeded later in the experiment after resubmission. All jobs, but two, completed successfully. The two jobs that failed had trouble connecting to the Ibis nameserver for unknown reasons. We also observe that there was negligible background load in all the clusters with exception of the Delft cluster.

4.7.3 IRunner experiment

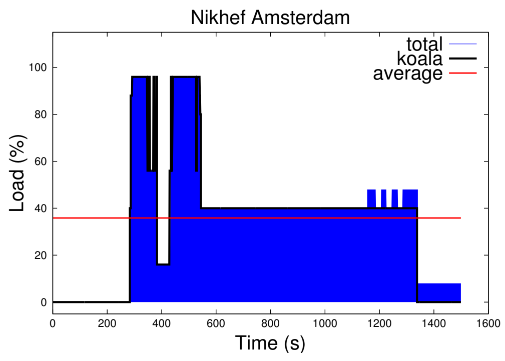
We have run the previous experiment in the exact same way with the IRunner to compare the two runners. This is not a completely fair comparison since grun also creates a sandbox and performs extra cleaning up actions after a run. Nevertheless we can observe the overhead the GRunner incurs in contrast to the IRunner. We again observe that the Amsterdam (VU) cluster was not used. All jobs finished successfully. We observe that there was considerable more background load in the Leiden en Amsterdam (Nikhef) clusters than in the previous experiment. We also see that despite the background load the IRunner was able to complete the experiment faster than the GRunner. This was to be expected since none of the jobs failed and consequently none of the jobs needed to be resubmitted. We conclude that the overhead of grun in the GRunner is not very large and is quite acceptable to live with if the user prefers the grun interface above writing a RSL description file.



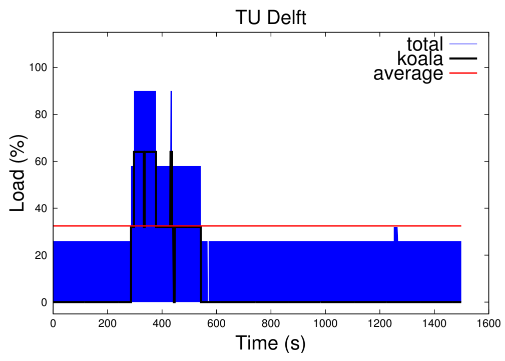
(a)



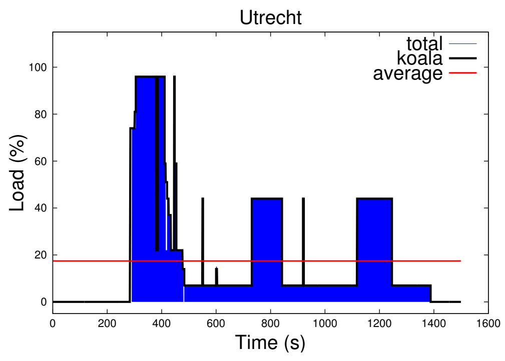
(b)



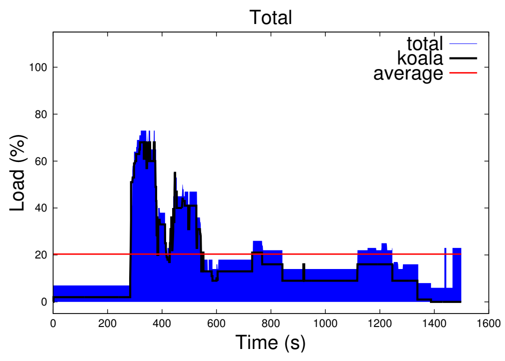
(c)



(d)

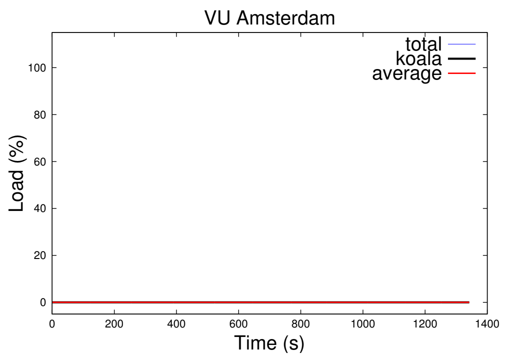


(e)

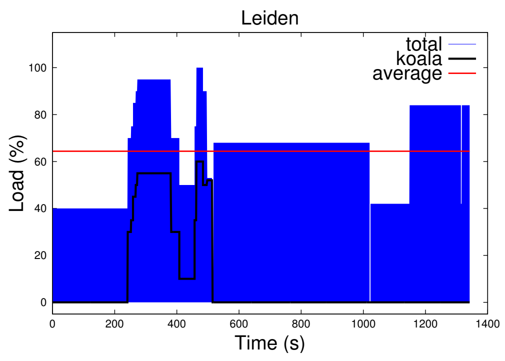


(f)

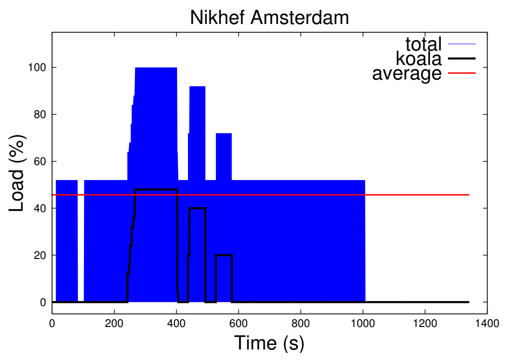
Figure 4.5: System utilisation for the GRunner experiment.



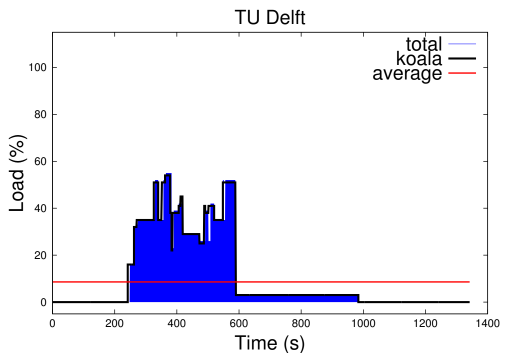
(a)



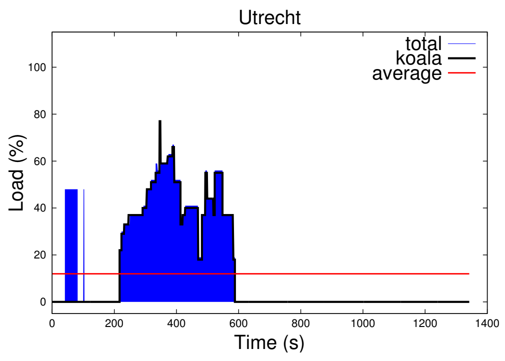
(b)



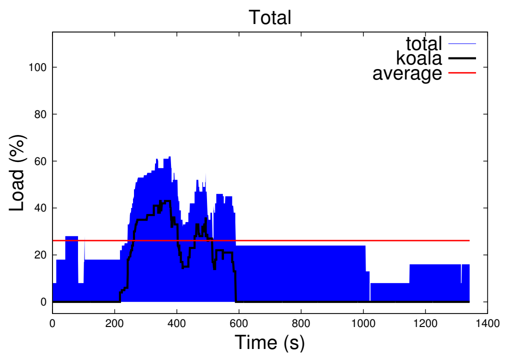
(c)



(d)



(e)



(f)

Figure 4.6: System utilisation for the IRunner experiment.

Chapter 5

Support for malleable jobs in Koala

In this chapter we turn our attention to the scheduling of another job type, malleable jobs, which are jobs that can change their size, i.e., their number of processors, during runtime. Malleable jobs are relatively new in the field of grids and not much research has been done on scheduling and running malleable jobs. The research that has been done focuses on simulation environments [26, 38, 7], or only considers a single cluster or single supercomputer [28]. Not all applications are malleable and even fewer applications can be adapted to become malleable. Ibis (see Section 4.1) however, does include support for malleable jobs. The Ibis runtime system is capable of detecting when processes leave/enter an Ibis run.

In this chapter we show how KOALA is able to run malleable jobs using the MRunner. The objective of constructing the MRunner is to show:

1. the possibility of supporting malleable jobs in KOALA
2. the relative ease with which KOALA is extended with new runners
3. the benefit of malleable jobs for system utilisation

In this chapter we will first define new job types in Section 5.1. In Section 5.2 we will motivate our choice for supporting malleable jobs by discussing some possible uses for malleable jobs. Our requirements for running malleable jobs are detailed in Section 5.3, followed by our design of the MRunner in Section 5.4. Our implementation of the MRunner is discussed in 5.5, followed by some experiments with the MRunner in Section 5.6.

5.1 Job types

In this section we extend the definition of job types already given in Section 2.2.1. Until now we only considered *rigid* jobs; fixed and non-fixed jobs are both *rigid*. Flexible jobs are often called *moldable* jobs in the literature, although flexible jobs are actually a generalisation of *moldable* jobs.

who / when	**at submission**	**during execution**
user	Rigid	Evolving
system	Moldable	Malleable

Table 5.1: Decisions about job size [19].

Jobs can be categorised by two properties:

- Can the job configuration, i.e., the composition of its components, change after submission?
- Is the job configuration set by the user or by the system?

Using these properties we can determine four job types [19]:

- The job configuration cannot be altered after submission
  - *Rigid*: There is only one job configuration, set by the user.
  - *Moldable*: The system chooses between different job configurations.
- The job configuration can be altered after submission
  - *Malleable*: The system can add and remove processors during execution.
  - *Evolving*: The application can request or relinquish processors during execution.

Table 5.1 shows an overview of different job types, and Figure 5.1 illustrates processor-time usage by different job types.

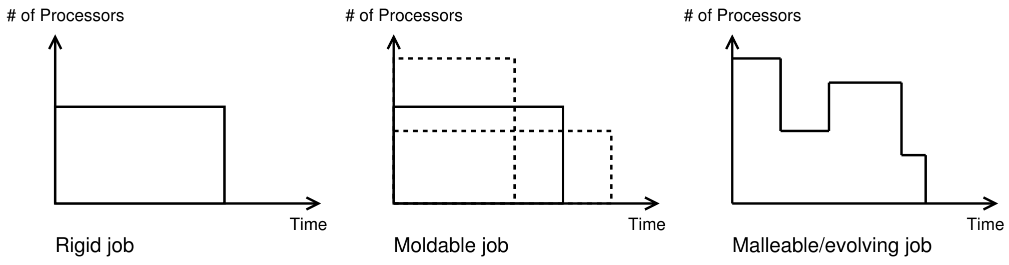


Figure 5.1: Processor-time diagrams of different job types.

Distinguishing between malleable and evolving jobs is not very useful. In the case of a malleable job, how does the system know when to stop supplying new resources to the malleable job? How useful is it for an evolving job to keep requesting new resources while there are none free? A combination of system and user initiated change might be the only feasible construction. Since evolving is actually a quite confusing term, we will refer to all kinds of jobs which can change during runtime as *malleable jobs*.

We will call increasing the number of claimed resources for a malleable job *growing* and releasing resources *shrinking*.

5.2 Possible uses of malleable jobs

Although malleable jobs are not very common in the grid, there are certainly some application types which could benefit from being able to change their configuration after they have started running. Besides this, it is very challenging to construct a scheduler for these jobs. In this section we discuss four possible uses of malleable jobs.

5.2.1 Satin

Satin is an extension of Ibis with Cilk like primitives. Satin has been developed to easily program divide-and-conquer style algorithms on wide-area systems. Some research has been done to make Satin programs fault-tolerant in a grid environment [55, 56]. The Ibis runtime system can handle crashing nodes as well as nodes which join the Ibis run after the application has started. In [55] an experiment has been done where a significant part of the processors were forcibly removed from the run and where, later on in a different location, some processors were added to the computation with a minimal loss of work that was already done.

Ibis, and Satin in particular, can therefore greatly benefit from being run as a malleable job.

5.2.2 Workflows

Workflows could be modelled as malleable jobs. Instead of releasing and requiring resources after every step in the workflow, the workflow could be seen as one big malleable job which shrinks and grows as necessary during the whole runtime of the workflow. One could even think of a 'conditional/dynamic workflow' where decisions about what to compute next are made according to resource availability.

Also simpler forms of applications where distinct phases of execution are present could use malleability. For instance, a simulation which after certain periods starts a visualisation phase, or real interactive jobs which need a different amount of resources after every user input.

5.2.3 Parameter sweep applications

In addition to co-allocated jobs users often want to run a lot of small single-processor applications. These applications are usually all the same but each run is done with different parameters, hence the name *parameter sweep application*. To avoid the overhead of acquiring grid resources every time for all these runs, one could write a KOALA runner which claims a number of processors and proceeds to run multiple of these small applications on them within one run.

Such a runner could act as a single malleable job, shrinking and growing whenever the system allows it. During its run it can execute as many of the smaller applications as possible. This behaviour comes close to a so-called *screen-saver job*, which is a job that takes up all idle cycles of a processor like Seti@Home. In this case it would not be hunting for idle CPU cycles but for idle processors. We are currently working on realising such a runner for KOALA.

5.2.4 Future grid applications

The nature of the applications currently executed on grids is related to the history of grid computing. In the times of parallel vector-machines, applications were usually modelled with the SIMD-principle (single instruction multiple data). The same application was run multiple times but with different data to operate on.

Nowadays the programmer has very powerful languages with which he can code complex runtime behaviour. First-year students, who participate in a Java programming course, are accustomed to starting threads as easily as declaring a new variable. Parallel programming seems to be lagging behind sequential programming. It is the author's opinion that parallel programming paradigms like used in MPI and PVM are somewhat outdated. The static, inflexible way of programming with, for instance MPI, should be replaced by the more dynamic, flexible style of programming with, for instance Ibis.

With new programmers being able to make use of such flexible languages the author predicts that more malleable applications may be written in the future. The grid should facilitate more flexible technologies so it can attract more types of applications than just the typical high performance computing applications. Sometimes it is not the demand which drives the development of new technologies but it is the technology which creates new demand. Being able to schedule malleable jobs may be the first step to let people start designing their applications with a more flexible paradigm in mind.

5.3 Requirements of the MRunner

This section will state the requirements for the MRunner in order to run malleable jobs, as well as state the necessary extensions to the co-allocator to enable the co-allocator to interact with the MRunner.

5.3.1 Specification of malleable jobs

The size of a malleable job is specified using three constraints:

- **Minimum:** the minimum number of processors a malleable job needs to be able to run, the job cannot shrink below this value. This value will often be one.
- **Maximum:** the maximum number of processors a malleable job can handle, allocating more processors would just waste resources.
- **Stepsize:** the number of processors the malleable job increases every time it grows.

A malleable job has a minimum and maximum number of resources it can handle as well as a restriction on how to *shrink* or *grow*. These three constraints define the *specification* of a malleable job. The *Stepsize* denotes the *step constraint*. A job could be constrained to always have an even number of resources or every increase should be quadratic in relation to the current number of resources. The MRunner is able to change the configuration of a malleable job while the job is running.

5.3.2 Initiative of change

Changing the size of a malleable job (shrinking or growing) can be initiated by:

- the user application; when a user application needs to grow or shrink then the change in size is directly related to the work the application is performing. For example, a computation can be in need of more processors before it can continue.

- the KOALA co-allocator; when the co-allocator decides that a malleable job has to shrink or grow this reflects the availability of free resources in the grid. For example, when there are not many free resources available anymore, the co-allocator requests the currently running malleable jobs to shrink to accommodate newly arriving jobs.

The MRunner should be able to handle requests for change in size from both. The co-allocator needs to know whether it is dealing with a malleable job and has to be aware of idle resources it can distribute among malleable jobs. The user application, the malleable job itself, needs to know how to communicate its requirements to the co-allocator. The MRunner coordinates the communication between the malleable job and the co-allocator.

5.3.3 Priority of change

Changing the size of a malleable job can be done at two levels;

- *voluntary*, a voluntary change means that the change does not have to succeed or does not necessarily have to be executed; the change is a recommended guideline.

- *mandatory*, a mandatory change has to be accommodated, because either the application cannot proceed without the change, or because the system is in direct need of reclaiming resources.

A mandatory increase of the number of resources needed by the user application means that the application cannot continue its work before new resources have been claimed by the application. A user application might also signal that it has more resources than it needs and consequently requests to shrink. Keeping the extra resources will not hurt the application, it will just waste resources. This request to shrink is therefore a voluntary change.

Conversely, when the co-allocator instructs a job to shrink because the system is low on free resources, it will be a mandatory change. When enough resources are available, the co-allocator can send a voluntary grow request; the malleable job is then offered new resources with a ‘take it or leave it’ policy.

5.3.4 Termination of malleable jobs

The last requirement of the MRunner is to be able to signal when the application is done. For conventional job types an application has finished when all its components have terminated; a malleable job, however, can still have work left to do when all its components have finished. The co-allocator needs to know when to stop offering resources to the malleable job.

5.3.5 Requirements of the co-allocator

The KOALA co-allocator has to be modified in order to support malleable jobs. It should be able to do the following:

- The co-allocator is aware of all running malleable jobs in the system and is able to distribute free resources over those malleable jobs. The co-allocator can distribute free resources fairly or with a bias; this bias can be based on the priority of the malleable jobs. An upper limit on assigning these resources can be set. For example, the co-allocator can only let malleable jobs grow if more than 20% of the resources are free. This limit prevents malleable jobs from taking all resources and consequently starving other jobs.
- The co-allocator should schedule economically; malleable jobs should not hog system resources but have to grow modestly by a small number of processors every step. In addition, it should try to place new components on clusters which already have some components of the malleable job to avoid letting the application communicate over the WAN connections. We call this *cluster-aware scheduling*. This is only a heuristic for selecting a cluster. New components can also be scheduled on a cluster which has no components of the malleable job present.
- The co-allocator can instruct running malleable jobs to shrink whenever resources are needed for new jobs. If a new job cannot be placed due to lack of free resources, the co-allocator tries to free up resources by ordering the malleable jobs to release a part of their resources. A graphic representation of this concept is shown in Figure 5.2. At first we see a malleable job slowly taking all resources in the system. When a new job enters the system it starts shrinking until the new job can be accommodated. Afterwards it incrementally reclaims more resources again.

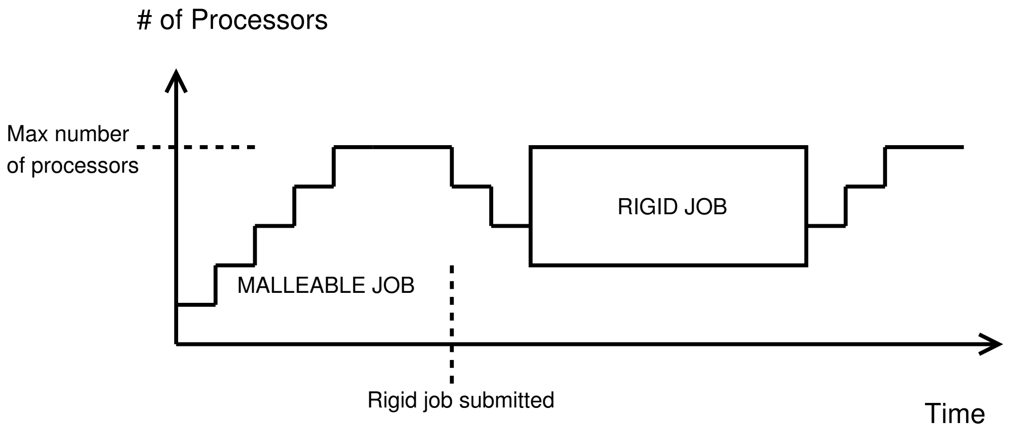


Figure 5.2: A malleable job has to make room for a ‘normal’ job.

5.4 The design of the MRunner

The support of malleable jobs in KOALA centres around the MRunner (the Malleable job Runner); we will discuss the design of the MRunner in this section. As a basis for the MRunner we take the simple KRunner. We describe how to model malleable jobs, how to control the shrinking and growing of malleable jobs, and the interaction between malleable jobs and the co-allocator.

5.4.1 The representation of malleable jobs

Conventional jobs consist of one or more components which can be fixed or non-fixed. Changing the sizes of these components during runtime is, to the writer's awareness, impossible with the current grid middleware (Globus) and is also impossible with the local schedulers (SGE). Once a job component starts executing it cannot gain or release resources anymore. To model the change in size of malleable jobs, we are left with adding and removing whole components to and from malleable jobs. Malleable jobs are thus expressed by a list of components. Malleable jobs grow by adding components to their list and shrink by removing a component, or combination of components, from their list. The drawback of this approach is that we cannot shrink a job by an arbitrary amount, but only by a combination of its running components.

The MRunner is responsible for starting and controlling all the components of a malleable job, it has a queue which contains all components. The co-allocator only has information on the total number of processors a malleable job has claimed; it is not aware of the list of components of malleable jobs.

5.4.2 Control of malleable jobs

The control over shrinking and growing lies with the MRunner. The MRunner is the 'liaison' between the user application and the co-allocator.

When a user application needs to grow, the application informs the MRunner. The MRunner then queries the co-allocator for a suitable location which has free resources and it will start a new component on that site. When a user application requests to shrink, the MRunner will abort a running job component or a combination of multiple running job components. The MRunner will try to abort the most recently submitted components to minimise the amount of work lost.

The MRunner also accommodates shrink and grow requests from the co-allocator. All shrink and grow requests go through the MRunner in order to simplify and control the malleability of jobs. This means that even when the co-allocator instructs a change, jobs are not immediately shrunk or grown. The co-allocator has to wait for the MRunner to handle the change and confirm the change to the co-allocator. The MRunner could even deny some requests or just not accept the offer of new resources depending on the state of the application. The co-allocator is informed of the actual claimed number of resources by the MRunner. An overview of the primitives is given in Figure 5.3.

It is up to the application itself to deal with the loss of resources, and consequently, work. Whenever a resource enters or leaves the application run, the application itself should provide mechanisms to distribute some work to that resource or to retrieve the

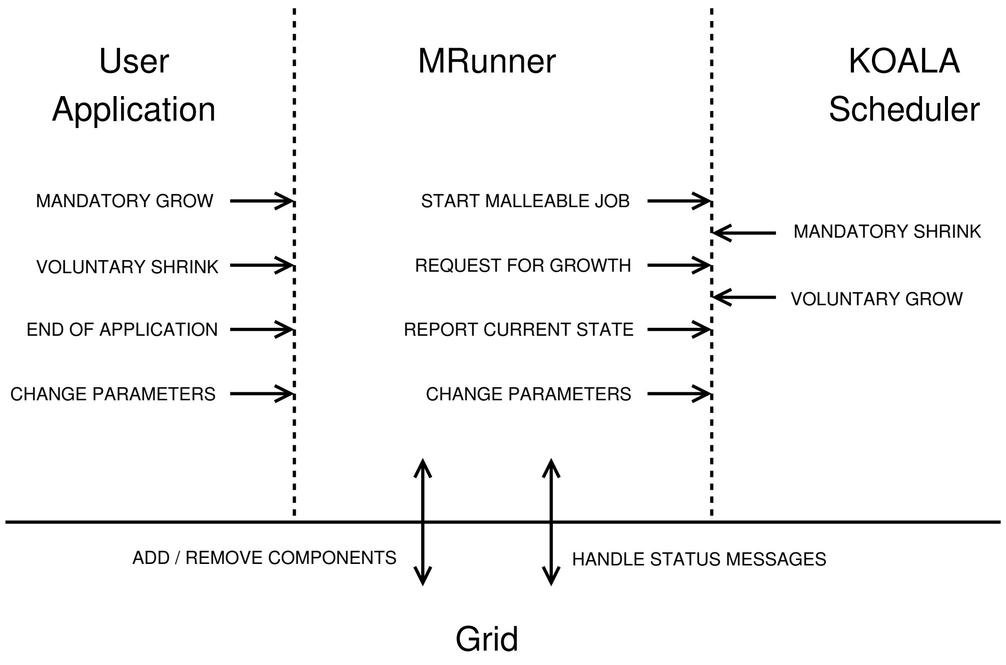


Figure 5.3: The design of the interfaces between components involved in running a malleable job.

work that resource has done. In [55], for instance, such a fault tolerant mechanism for Satin jobs has been constructed.

5.4.3 Phases of malleable jobs

When malleable jobs are submitted they are treated by the co-allocator as normal jobs, with the same priorities normal jobs can have. They cannot grow or shrink until all their initial components are running. We call this the *startup phase*. In the startup phase malleable jobs are treated like any other job. After all components have been successfully submitted we enter the *malleable phase*. In the malleable phase jobs can start growing and shrinking until they are finished. Figure 5.4 shows the phases of a malleable job.

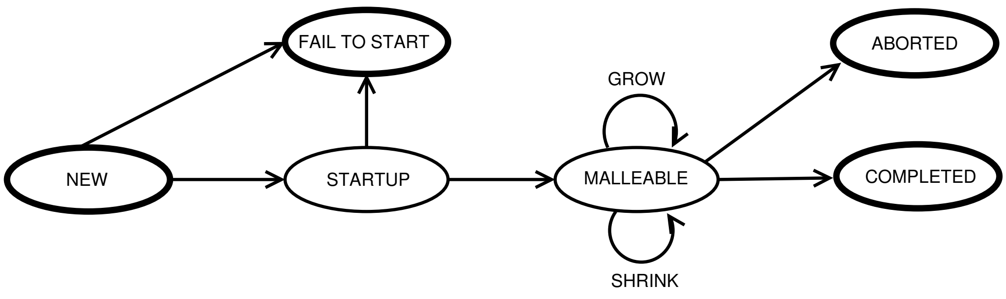


Figure 5.4: The phases of a malleable job.

5.4.4 Submission protocol of malleable jobs

Growing and shrinking a job can be done by adding a new component to the malleable job. This new component has the size specified in the step constraint and is submitted to the co-allocator as a new stand-alone job. After the new component has entered the run queue it is merged with the already running malleable job. In contrast to a co-allocated job, when a component fails it does not mean that the computation will be incorrect and therefore the job does not need to be aborted. We just let that component fail and re-request a new component, or if growing is mandatory we wait until new resources become available.

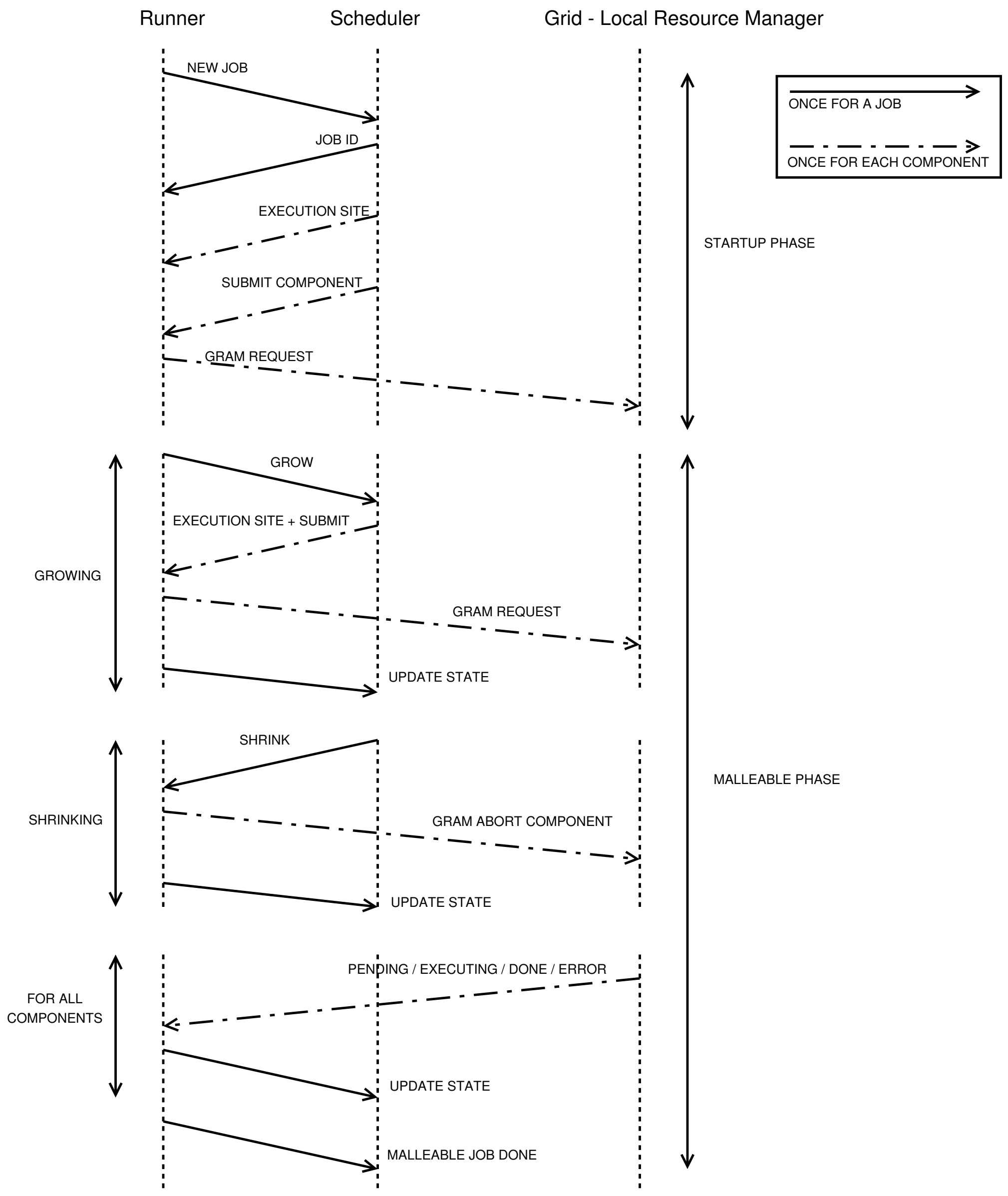


Figure 5.5: Extensions to the submission protocol for malleable jobs.

Figure 5.5 shows the extensions to the submission protocol; this figure should be compared with Figure 3.5 which depicts the protocol of the KRunner. The submission mechanism, Globus GRAM, has not changed. The two parts marked *growing* and *shrinking* can be done repeatedly. The part marked *for all components* depicts that during shrinking and growing all running components still produce status messages which are processed by the MRunner.

5.5 The implementation of the MRunner

In this section we will discuss implementation details for the MRunner. A full implementation of the MRunner has not yet been done. Optimal scheduling of malleable jobs entails a lot of research and the communication between the runner and the user application is also a part that needs to be studied more. We did however make a prototype implementation of the MRunner. In this section we will first detail the restrictions we place on the MRunner, followed by details about the implementation of the job description file and the submission protocol.

5.5.1 Restrictions of the MRunner

We have made some restrictions and assumptions to be able to construct the MRunner, but still the benefit and feasibility of scheduling malleable jobs can be shown.

- **User application initiated change** We removed all communication between the runner and the user application. Although this is a very important factor for correct execution of malleable jobs, it is not strictly necessary to be able to implement malleability. All changes are thus initiated by the co-allocator. We assume the application is able to handle newly arriving and leaving resources.

- **Termination of malleable jobs** Without communication between the user application and the MRunner, we cannot know when the computation is done. We simulate the termination of an application by counting the number of components which have completed successfully. After a predefined number of completed components we decide that the application has performed all its work. This number can be specified by the user. The work that an application computes is thus modelled as a number of components that need to finish. This adequately mimics the behaviour of, for example, divide-and-conquer applications, where the total work is recursively split into subjobs. When those subjobs finish, they report their results back to their parent subjobs. All the results are finally aggregated by the root job.

  Other applications might not be modelled very well by this approach. A possible scenario could be a MPI job which starts running with a number of processors, increases that number once, and subsequently keeps all its resources until it has finished. For this scenario we need communication between the user application and the MRunner.

- **Specification of malleable jobs** It is not possible to change the specification of a malleable job. The minimum, maximum, and step constraint should be able to change during the run of a malleable job, see Section 5.3. The absence of this functionality will be a nuisance to application programmers but does not affect the goal of our prototype.

The step constraint is assumed to be a constant. Although the number with which to increase and decrease the job should be able to vary, we chose to make it a constant for simplicity.

- **Malleable job scheduling** No cluster-aware scheduling is done since this would mean rewriting large parts of the co-allocator. The malleable job can take up all resources in a cluster, and we do not check for a limit on utilisation.

- **File staging** We do not support file staging to avoid many complications this would bring. Only basic input and output (stdin and stdout) are handled.

5.5.2 The job description file for malleable jobs

To enable users to submit malleable jobs, we have extended the RSL specification with some extra attributes. These extra RSL attributes are only applicable to job description files for malleable jobs and these attributes are not defined by GRAM. The job description file only uses these extra attributes for the specification of a malleable job; the normal RSL specification still applies for defining normal components.

In addition to specifying normal job components, the user specifies two ‘special’ components which contain the details of the malleable job. These two special components are not submitted; they act as placeholders for the parameters of the job. One of these components gives the specification, i.e., the parameters, of a malleable job, the other defines the RSL which is submitted when a malleable job grows. The new RSL attributes are:

- `minimum:` the minimum number of processors a malleable job needs to run, the job cannot shrink below this value. This will usually be just one.

- `maximum:` the maximum number of processors a malleable job can handle, allocating more would just be wasting resources.

- `stepsize:` the number of processors a malleable job increases with every growth. This is needed for the co-allocator to determine if there is enough free space for a job to grow.

- `work:` the number of successfully completed components before a job has finished.

- `malleable:` this attribute can have two values: `specification` or `step`. The `malleable` attribute is used to define the two ‘special’ components and to distinguish them from normal components.

The normal components are started in the startup phase, the `step` component is submitted every time a malleable job grows in the malleable phase. An example of a job description file for a malleable job is given in Figure 5.6. The first two components are normal components, the third component defines the parameters of the malleable job, the last component specifies the RSL which is submitted as a new job when a job grows.

```
+
(
 &( directory = "/home/wlammers/experiments" )
  ( executable = "/home/wlammers/experiments/malleable_job" )
  ( maxWallTime = "20" )
  ( label = "subjob 0" )
  ( count = "10" )
  ( arguments = "200" )
)
(
 &( directory = "/home/wlammers/experiments" )
  ( executable = "/home/wlammers/experiments/malleable_job" )
  ( maxWallTime = "20" )
  ( label = "subjob 1" )
  ( count = "10" )
  ( arguments = "200" )
)

(
 &( malleable = "specification" )
  ( minimum = "4" )
  ( maximum = "70" )
  ( minimumStepSize = "4" )
  ( work = "40" )
)
(
 &( malleable = "step" )
  ( directory = "/home/wlammers/experiments" )
  ( executable = "/home/wlammers/experiments/malleable_job" )
  ( maxWallTime = "20" )
  ( label = "subjob 0" )
  ( count = "4" )
  ( arguments = "10" )
)
```

Figure 5.6: An example of a job description file for a malleable job.

5.5.3 The implementation of the submission protocol

The MRunner inherits from the same runner codebase library as the KRunner does. This is the normal starting point for extending KOALA with a new runner. We were able to reuse much of the code that was used in the KRunner. We extended the submission protocol with grow and shrink messages for both the co-allocator and the MRunner. A separate queue for holding malleable jobs was added to the co-allocator. This queue is scanned round-robin to see if malleable jobs can grow; it has the lowest priority possible in KOALA. In addition to extending the protocol we introduced the startup and malleable phase to the MRunner.

All other runners leave processing GRAM messages up to the shared runners library. For the MRunner we needed to bypass this shared code, otherwise all components would be aborted, or restarted, when an error occurs. All components being finished is **not** an indication of job completion anymore. The GRAM notifications which signal job completion are therefore also handled by the MRunner. In the startup phase the shared runners library handles the GRAM messages. In the malleable phase we override the GRAM error and completion message to basically do nothing except update the internal administration about the job configuration. When a new component is successfully submitted, it is merged into the job component list of the malleable job in the MRunner and the co-allocator is informed that the job has grown.

5.6 Experiments

We have done three experiments with the MRunner. This section is organised as follows. First we describe the application used in the experiments, followed by a discussion of the results of each experiment separately.

5.6.1 Test application

The user application in our experiments is a simple Ibis job. The application detects whether new components have joined the Ibis run. If it detects a new component, or if it detects that a component has left, the application sends a message to all running components. The message is an irrelevant ”Hello (or Bye) Ibis instance number X”. In addition to detecting new Ibis instances the application just sleeps, the sleeping period can be set by the user at the start of the application. The sleeping period models the length of the computation of a component. The sleeping period for all experiments is set to 200 seconds. The application itself is not very useful but can still be used to model how a malleable job works. The Ibis application starts every instance with the so-called `worldmodel` property set to *open*. When the worldmodel is open, new Ibis instances can join the computation at any time. If the worldmodel is closed, then the Ibis application needs to know the number of processes that will be participating in the run. The application will block until that number of Ibis instances have registered themselves at the nameserver.

5.6.2 Experiment 1

With the first experiment we want to show that malleable jobs, with a small enough step size, can improve the system utilisation. For this experiment we submit one malleable job to a single cluster (Utrecht). The minimum number of processes for our malleable job is 1 and the maximum is 80. The cluster in Utrecht has 31 nodes, which corresponds to 62 processors. The maximum of the malleable job is set higher than the total number of processors so we can push system utilisation to 100%. The number of components that need to be completed successfully before the job is terminated is set to 40. A new component of 4 processors is submitted whenever our malleable job grows. In the startup phase we submitted only one component, which is also of size 4. Figure 5.7 shows the utilisation of the cluster in Utrecht during this experiment.

The filled surface illustrates the total load of the cluster, the thick black line depicts the contribution to the load of the malleable job. The straight grey line is the average load of the whole experiment which was 59%. We see that the load of the system is completely generated by the malleable job; during the experiment no other users submitted jobs to this cluster, so we observe no ‘background load’. At the start of the experiment we see the malleable job growing to almost 100% utilisation; because two nodes were experiencing problems, the load does not reach the full 100%. Even in a grid environment as small as the DAS we regularly have faulty nodes. Nevertheless, the malleable job is able to claim all free processors that are not having problems. After approximately 200 seconds we see a drop in the load which shows that some

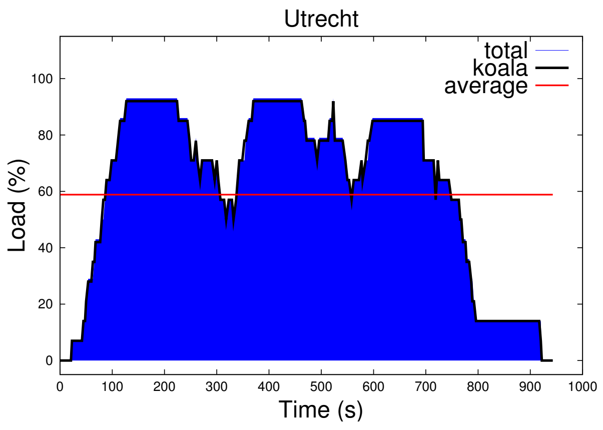


Figure 5.7: The improvement of the utilisation with malleable jobs in Experiment 1.

components have finished. Because all components run for approximately the same time there is a period where a lot of components are finishing. After a short period we see the job growing back to full utilisation again. A similar problem is observed after approximately 450 seconds. The time in between two peaks is the overhead of submitting, and subsequently, scheduling, placing, and claiming of new components. The period of low utilisation after 800 seconds of the experiment is caused by a few last components which are executing the last pieces of work.

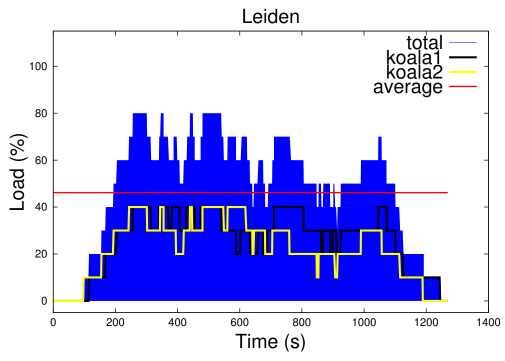
The experiment clearly shows that malleable jobs are able to improve system utilisation significantly. The load did not drop below 50% and the average load was 59%. The bottleneck lies in the speed with which we can submit new components.

5.6.3 Experiment 2

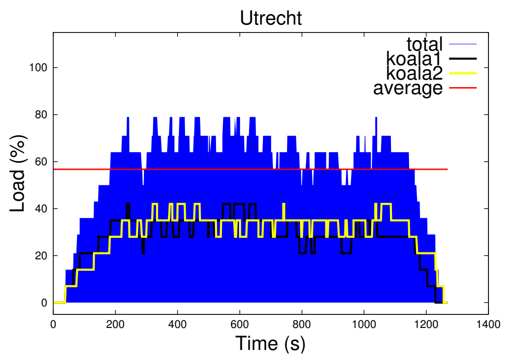
In the second experiment we want to show that KOALA distributes free resources fairly among running malleable jobs. We use two clusters, the Utrecht cluster the Leiden cluster. We start two malleable jobs simultaneously, both the jobs are the same as used in the previous experiment. The parameters for the two jobs are exactly the same as in the first experiment. This shows a situation in which two malleable jobs are competing for resources and how KOALA tries to distribute available resources fairly among the two. The results are shown in Figure 5.8.

Figure 5.8(a), 5.8(b), and 5.8(c) show the load of the Leiden cluster, the Utrecht cluster, and the total average load of all clusters in the DAS respectively. Since two clusters can only generate up to a load of approximately 40% of the total DAS load, the vertical axis in 5.8(c) ranges from 0 to 50. The black and white line each represent one of the malleable jobs. The load of the separate clusters are similar to the load as experienced in Experiment 1.

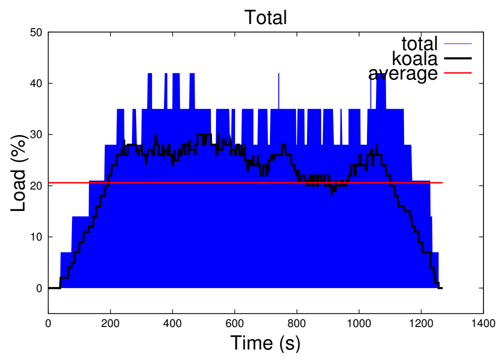
We see that resources are distributed equally among both jobs and both jobs are thus contributing equally to the total load of the clusters. In Figure 5.8(c) we see that running more malleable jobs together decreases the fluctuating behaviour of the load, i.e., there is a more constant utilisation instead of many highs and lows.



(a)



(b)



(c)

Figure 5.8: Fair growth among malleable jobs in Experiment 2.

5.6.4 Experiment 3

In the last experiment we want to show a running malleable job that is ordered to shrink and make room for a rigid job. The malleable job has the same parameters as in the previous experiments, with exception of its minimum, which is set to 0. We submit one instance of our malleable job to a single cluster (the Leiden cluster), which has 40 processors. When the malleable job has many components running we submit a second job to the same cluster. The second job is a rigid fixed job of size 40 and runs for approximately 120 seconds. The rigid job itself is a dummy application which sleeps during its runtime. Because the rigid job requires all available processors in the cluster, the malleable job will have to shrink to accommodate the fixed job. Figure 5.9 illustrates Experiment 3.

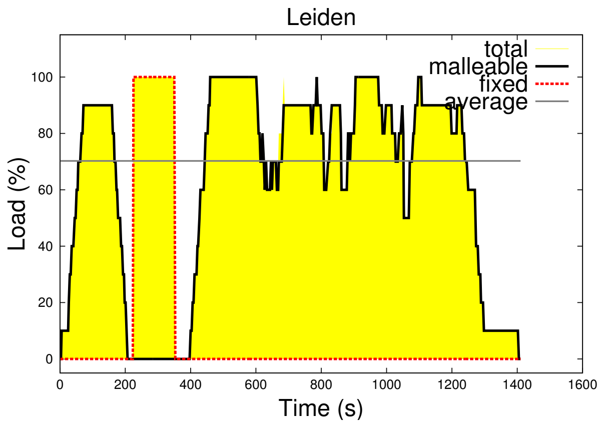


Figure 5.9: A malleable jobs makes space for a rigid fixed job in Experiment 3.

At first our malleable job grows as much as it can, but after 200 seconds it starts to shrink rapidly, because we have submitted the second job which is placed on the same cluster. The co-allocator repeatedly instructs our malleable job to release its processors to make space for the rigid job. The second big peak in the figure shows the execution of the rigid job. After this job has finished, the malleable jobs starts growing towards maximal utilisation again.

This experiment shows that malleable jobs do not hog system resources. Our malleable job in this experiment is an extreme case of course since it requires all processors of the cluster, but it adequately shows the flexibility of malleable jobs.

Chapter 6

Conclusion and Future Work

6.1 Conclusions

In this thesis we have presented the design and implementation of multiple submission procedures for applications to a grid. These submission procedures have been implemented as a runners in the KOALA grid scheduler. We have presented the design and implementation of the runner architecture, the communication protocol between KOALA's co-allocator and its runners, and the submission mechanisms to a grid. We have created four of KOALA's runners. The KRunner is used for relatively simple jobs. To submit jobs which use a special wide area communication library called Ibis, we created two runners called the GRunner and the IRunner, based on the submission tool called grun and based on the KRunner, respectively. We have also developed a prototype implementation of the MRunner, which is able to submit malleable jobs, which can change their size during runtime.

We have shown through our experiments that all the runners are reliable and that they make the submission of jobs to a grid considerably easier than existing tools by supporting non-fixed jobs and by supporting KOALA's fault tolerance mechanisms. The modular structure of KOALA allows us to easily add runners to KOALA. We have shown the encapsulation of the grun submission tool by the GRunner, which adds support for non-fixed jobs and fault tolerance to Ibis applications. The overhead of encapsulating grun has been shown by a comparison with the IRunner. We have also shown that KOALA is able to run malleable jobs which use can use any idle processors in the system, thus improving system utilisation without hindering the execution of rigid jobs.

KOALA was designed to submit co-allocated jobs to a grid, these jobs can claim a large number of processors and may take a long time to execute. We encountered three problems while running our experiments. First it became clear that KOALA's runners are not designed to handle a very large number of jobs. When running more than approximately two hundred jobs from one submission site, the file server performance strongly decreased. The reason is that each instance of a runner requires a separate Java virtual machine, which stresses the submission machine severely. KOALA's co-allocator itself has no trouble with handling more than two hundred jobs though.

Second, since the DAS is used exclusively as a research grid, it is not always heavily loaded. Periods of much activity typically occur at night and when many student lab courses are scheduled. We observed on the few occasions when the system was heavily loaded, that KOALA was ‘too nice’, i.e., it did not submit any jobs while the system load was high. Because almost all processors were claimed, jobs started to queue at the local resource managers. KOALA always tries to look for idle processors and consequently does not add its jobs to the queues of the local schedulers.

The last problem is that the number of network connections increases rapidly when running many separate runners. Each runner opens multiple network connections to communicate with the co-allocator, to cooperate with the GRAM job managers, and possibly to transfer multiple input files and output files. Especially when running jobs with many components, we observed that under heavy load during the stage out phase, when all output files are transferred to the submission site, some connections were being dropped.

6.2 Future Work

We can extend KOALA in many different ways and we can add many different runners to KOALA. We suggest only some of the possibilities for future work.

- It is possible within the RSL job description file to specify a limited sandbox for an application to run in. It is now up to the user to specify the sandbox, which makes it unsafe and laborious. An automatic sandbox for executing the jobs could be a good extension to the functionality of the runners.

- A solution to decrease the number of network connections while running many jobs can be to combine all file transfers and keep a pool of open GridFTP connections. A fixed number of open connections could then be maintained.

- For running large batches of jobs, one could create a multirunner or a batchrunner, which is a single Java virtual machine but runs many jobs. Only one process, i.e., one JVM, is then needed which handles all GRAM connections. One could even let this runner implement a form of application level scheduling by running more than one job on a node for the period that it has claimed that node.

- For the MRunner, many issues need to be addressed:

  - The MRunner can only handle system-initiated job-size changes and needs to be extended to be able to handle user-initiated changes. This means that a connection has to be designed between the MRunner and the user application.
  - In addition to user initiated change, the MRunner needs to query the user application to know when it has finished. This should also be considered when designing the protocol between the MRunner and the user application.
  - The effect of different priorities of scheduling malleable jobs in combination with rigid jobs needs to be studied in detail. A scheduling policy like the cluster-aware scheduling policy described in this thesis needs to be implemented.

- The MRunner also should be extended with support for an efficient way to transfer input files for job components of malleable applications.

Bibliography

[1] *Berkeley Open Infrastructure for Network Computing*. `http://boinc.berkeley.edu`.

[2] *Ibis: Efficient Java-based Grid Computing*.

[3] *The Distributed ASCI Supercomputer 2 (DAS-2)*.

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

[5] *The Portable Batch System, OpenPBS*. `www.openpbs.org`.

[6] J. Abawajy and S. Dandamudi. Parallel Job Scheduling on Multicluster Computing Systems. *IEEE International Conference on Cluster Computing*, pages 11–18, 2003.

[7] J. Blazewicz, M. Machowiak, G. Mounie, and D. Trystam. Approximation Algorithms for Scheduling Independent Malleable Tasks. *7th International Euro-Par Conference Manchester on Parallel Processing*, 2150:191–197, 2001.

[8] J. L. Bosque and L. P. Perez. HLogGP: A New Parallel Computational Model for Heterogeneous Clusters. *IEEE International Symposium on Cluster Computing and the Grid*, pages 403–410, 2004.

[9] J. L. Bosque and L. P. Perez. Theoretical Scalability Analysis for Heterogeneous Clusters. *IEEE International Symposium on Cluster Computing and the Grid*, pages 285–292, 2004.

[10] A. Bucur and D. Epema. An Evaluation of Processor Co-Allocation for Different System Configurations and Job Structures. *14th Computer Architecture and High Performance Computing*, pages 195–203, 2002.

[11] A. Bucur and D. Epema. Priorities among Multiple Queues for Processor Co-Allocation in Multicluster Systems. *36th Simulation Symposium*, pages 15–27, 2003.

[12] A. Bucur and D. Epema. The Maximal Utilization of Processor Co-Allocation in Multicluster Systems. *Parallel and Distributed Processing Symposium*, page 10 pp, 2003.

[13] A. Bucur and D. Epema. The Performance of Processor Co-Allocation in Multicluster Systems. *Cluster Computing and the Grid*, pages 302–309, 2003.

[14] A. R. Butt, R. Zhang, and Y. C. Hu. A Self-Organizing Flock of Condors. *Supercomputing Conference*, 2003.

[15] K. Czajkowski, S. Fitzgerald, I. Foster, and C. Kesselman. Grid Information Services for Distributed Resource Sharing. *10th IEEE International Symposium on High Performance Computing*, pages 181 – 194, 2001.

[16] K. Czajkowski, I. Foster, N. Karonis, C. Kesselman, S. Martin, W. Smith, and S. Tuecke. A Resource Management Architecture for Metacomputing Systems. *Proc. IPPS/SPDP*, pages 62–82, 1998.

[17] K. Czajkowski, I. Foster, and C. Kesselman. Resource Co-Allocation in Computational Grids. *8th IEEE International Symposium on High Performance Distributed Computing*, pages 219 – 228, 1999.

[18] T. Eymann, M. Reinicke, O. Ardaiz, P. Artigas, F. Freitag, and L. Navarro. Decentralized Resource Allocation in Application Layer Networks. *3rd IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 645 – 650, 2003.

[19] D. G. Feitelson and L. Rudolph. Toward Convergence in Job Schedulers for Parallel Supercomputers. *Job Scheduling Strategies for Parallel Processing JSSPP*, 1162:1–26, 1996.

[20] I. Foster. What is the Grid? A Three Point Checklist. *GridToday*, 2002.

[21] I. Foster and C. Kesselman. Computational Grids. *The Grid: Blueprint for a Future Computing Infrastructure*, 1998.

[22] I. Foster, C. Kesselman, J. M. Nick, and S. Tuecke. The Physiology of the Grid. 2002. `http://www.globus.org/research/papers/ogsa.pdf`.

[23] I. Foster, C. Kesselman, and S. Tuecke. The Anatomy of the Grid. *Lecture Notes in Computer Science*, volume 2150, 2001. `http://citeseer.ist.psu.edu/foster01anatomy.html`.

[24] D. Gunter, B. Tierney, K. Jackson, J. Lee, and M. Stoufer. Dynamic Monitoring of High-Performance Distributed Applications. *High Performance Distributed Computing*, pages 163–170, 2002.

[25] E. Heymann, M. Senar, E. Fernandez, A. Fernandez, and J. Salt. Managing MPI Applications in Grid Environments. volume 3165, pages 42–50, 2004.

[26] J. Hungershofer. On the Combined Scheduling of Malleable and Rigid jobs. *16th Symposium on Computer Architecture and High Performance Computing (SBAC-PAD*, pages 206–213, 2004.

[27] A. Iamnitchi and I. Foster. A Peer-to-Peer Approach to Resource Location in Grid Environments. *High Performance Distributed Computing*, page 419, 2002.

[28] L. V. Kale, S. Kumar, and J. DeSouza. A Malleable-Job System for Timeshared Parallel Machines. *2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid)*, pages 230–238, 2002.

[29] N. Karonis, B. Toonen, and I. Foster. MPICH-G2: A Grid-Enabled Implementation of the Message Passing Interface. *Journal of Parallel and Distributed Computing (JPDC)*, 63:551–563, 2003. No. 5.

[30] Y.-S. Kim, J.-L. Yu, J.-G. Hahm, J.-S. Kim, , and J.-W. Lee. Design and Implementation of an OGSI-Compliant Grid Broker Service. *IEEE International Symposium on Cluster Computing and the Grid*, pages 754–761, 2004.

[31] K. Kumar, A. Agarwal, and R. Krishnan. Fuzzy Based Resource Management Framework for High Throughput Computing. *IEEE International Symposium on Cluster Computing and the Grid*, pages 555–562, 2004.

[32] W. Lammers. Resource management in grids, 2004.

[33] C. A. Lee, J. Stepanek, R. Wolski, C. Kesselman, and I. Foster. A Network Performance Tool for Grid Environments. *Supercomputing Conference*, pages 1–16, 1999.

[34] C. Liu, L. Yang, I. Foster, and D. Angulo. Design and Evaluation of a Resource Selection Framework for Grid Applications. *High Performance Distributed Computing*, pages 63–72, 2002.

[35] H. Mohamed and D. Epema. Experiences with the KOALA Co-Allocating Scheduler in Multiclusters. *IEEE/ACM Int*, 5th, 2005. Cardiff.

[36] H. Mohamed and D. Epema. The Design and Implementation of the KOALA Co-Allocating Grid Scheduler. *European Grid Conference*, 2005. Amsterdam.

[37] H. Mohammed and D. Epema. An Evaluation of the Close-to-Files Processor and Data Co-Allocation Policy in Multiclusters. *IEEE International Conference Cluster Computing*, 2004.

[38] G. Mounie, C. Rapine, and D. Trystam. Efficient Approximation Algorithms for Scheduling Malleable Tasks. *11th Efficient Approximation Algorithms for Scheduling Malleable Tasks*, pages 23–32, 1999.

[39] M. S. Muller, E. Gabriel, and M. M. Resch. A software development environment for Grid computing. *Concurrency and Computation: Practice and Experience*, 14:1543–1551, 2002.

[40] P. Neophytou, N. Neophytou, , and P. Evripidou. Net-dbx-G: A Web-Base Debugger of MPI Programs over Grid Environments. *IEEE International Symposium on Cluster Computing and the Grid*, pages 35–42, 2004.

[41] A. Othman, P. Dew, K. Djemame, and I. Gourlay. Adaptive Grid Resource Brokering. *Cluster Computing*, pages 172–179, 2003.

[42] S.-M. Park and J.-H. Kim. Chameleon: A Resource Scheduler in A Data Grid Environment. *3rd IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 258 – 265, 2003.

[43] R. Raman, M. Livny, and M. Solomon. Policy Driven Heterogeneous Resource Co-Allocation with Gangmatching. *12th IEEE International Symposium on High Performance Computing*, pages 1–10, 2003.

[44] K. Ranganathan and I. Foster. Decoupling Computation and Data Scheduling in Distributed Data-Intensive Applications. *11th IEEE International Symposium on High Performance Distributed Computing*, pages 352 – 358, 2002.

[45] H. Shan and L. Oliker. Job Superscheduler Architecture and Performance in Computational Grid Environments. *Supercomputing Conference*, pages 1–15, 2003.

[46] J. Sinaga, H. Mohammed, and D. Epema. A Dynamic Co-Allocation Service in Multicluster Systems. *10th Workshop on Job Scheduling Strategies for Parallel Processing*, 2004.

[47] J. M. P. Sinaga. A Dynamic Co-Allocation Service in Multicluster Systems. Master's thesis, 2004.

[48] W. Smith, V. Taylor, and I. Foster. Using Run-Time Predictions to Estimate Queue Wait Times and Improve Scheduler Performance. *5th Workshop on Job Scheduling Strategies for Parallel Processing*, 1659:202–219, 1999.

[49] V. Subramani, R. Kettimuthu, S. Srinivasan, and P. Sadayappan. Distributed Job Scheduling on Computational Grids using Multiple Simultaneous Requests. *11th IEEE International Symposium on High Performance Computing*, pages 359 – 366, 2002.

[50] D. Thain and M. Livny. The Ethernet Approach to Grid Computing. *12th IEEE International Symposium on High Performance Distributed Computing*, pages 138–147, 2003.

[51] S. S. Vadhiyar and J. J. Dongarra. A Metascheduler for the Grid. *11th IEEE International Symposium on High Performance Computing*, pages 343 – 351, 2002.

[52] 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*, 16:1–29, 2002.

[53] L. Wang, W. Cai, and B.-S. Lee. Resource Co-Allocation for Parallel Tasks in Computational Grids. *International Workshop on Challenges of Large Applications in Distributed Environments*, pages 88 – 95, 2003.

[54] G. Wasson, N. Beekwilder, M. Morgan, , and M. Humphrey. OGSI.NET: OGSI-compliance on the .NET framework. *IEEE International Symposium on Cluster Computing and the Grid*, pages 648–655, 2004.

[55] G. Wrzesinska, J. Maassen, and H. E. Bal. A Simple and Efficient Fault Tolerance Mechanism for Divide-and-Conquer Systems. *CCGrid 2004*, 2004.

[56] G. Wrzesinska, R. V. van Nieuwpoort, J. Maassen, and H. E. Bal. Fault-Tolerance, Malleability and Migration for Divide-and-Conquer Applications on the Grid. *ipdps2005*, 2005.

[57] H.-J. Zhang, Q.-H. Li, and Y.-L. Ruan. Resource Co-Allocation via Agent-Based Coalition Formation in Computational Grids. *2nd International Conference on Machine Learning and Cybernetics*, pages 1936–1940, 2003.

[58] X. Zhang, J. L. Freschl, and J. M. Schopf. A Performance Study of Monitoring and Information Services for Distributed Systems. *12th IEEE International Symposium on High Performance Computing*, pages 270 – 281, 2003.

[59] X. Zhang and J. M. Schopf. Performance Analysis of the Globus Toolkit Monitoring and Discovery Service, MDS2. *Performance, Computing, and Communications*, pages 843–849, 2004.

[60] D. Zhou and V. Lo. Cluster Computing on the Fly: Resource Discovey in a Cycle Sharing Peer-to-Peer System. *Cluster Computing and the Grid*, pages 66–73, 2004.