

A Dynamic Co-allocation Service in Multicluster Systems

Jove Meldi P. Sinaga

April 2004

A Dynamic Co-allocation Service in Multicluster Systems

THESIS

to obtain the title of
Master of Science in Technical Informatics
at Delft University of Technology,
Faculty of Electrical Engineering, Mathematics, and Computer Science
Parallel and Distributed Systems Group.

by

Jove Meldi P. Sinaga

April 2004

Graduation Data

Author : Jove Meldi Priyatama Sinaga
Title : A Dynamic Co-allocation Service in Multicluster Systems
Graduation date : March 29, 2004

Graduation Committee:
Prof. dr. ir. H.J. Sips (voorzitter) Delft University of Technology
dr. ir. D.H.J. Epema Delft University of Technology
ir. F.R. van der Vlugt Delft University of Technology

Abstract

In multicluster systems, and more generally in grids, jobs may require co-allocation, i.e., the simultaneous allocation of resources such as processors in multiple clusters to improve their performance. In previous work, processor co-allocation have been studied through simulations. Here, we extend the work with the design and implementation of a dynamic processor co-allocation service. While a co-allocation mechanism has been implemented for some years in DUROC component of the Globus Toolkit, DUROC has some shortcomings in that it does not do any resource brokering nor does it provide complete fault tolerance to handle job submission and completion failures. In this work, we add these two elements in the form of a software layer on top of DUROC, by emphasizing the resource brokering element so that the co-allocation service can detect the states of processors before deciding whether or not to submit job requests. We have performed experiments showing that our co-allocation service works correctly as designed.

Preface

This report is the documentation of my Master's thesis at the Parallel and Distributed Systems Group, Faculty of EEMCS, Delft University of Technology.

On this occasion, I would like to thank my supervisor, dr. ir. D.H.J.Epema, for his patience, help, and broad insight to direct me in this research. Also, I want to thank Hashim H. Mohammed for his contribution in some important ideas, and programming things. I am grateful for the complete facility and resources to support this work.

I also thank my parents, brothers, and sisters who gave me support and encouragement during this work. I would not forget my brethren Alex Priyo Pratomo, Erich Pessiwarisa, Teddy Pardamean Sitorus, Menas Wibowo, Joshua Sitohang, Daniel Panjaitan, who also supported me. I would also appreciate support from my brethren in the Lord: Gerrit Benschop, Pauline, Robert and Renate Holms, Andrew and Pearl Barendse, Anton and Marie Janssen, Ammeret and Christien, Igor, and Laura.

Especially I would thank for my beloved, Weislina, who always gives her love and support to me. You are so special to me.

Above all, I would thank the God Almighty for His grace and wisdom so that I can finish my study well in this university. Thank You for giving me the opportunity to know and serve You.

Many things I have learned during this study. I realize those are important to mold me to get better in the future and contribute good things to others.

Delft, April 2004
Jove Meldi P. Sinaga

Contents

1	Introduction	1
2	Grid Computing in the Globus Environment	3
2.1	The Grid Architecture	3
2.2	Globus Basic Services	4
2.2.1	Security Service	4
2.2.2	Data Management	5
2.2.3	Information Service	5
2.2.4	Resource Management	7
	Resource Specification Language	7
3	The Concept of Co-allocation in Grids	9
3.1	Co-allocation Mechanism	9
3.1.1	How Globus DUROC works	9
	Allocation Period	10
	Configuration Period	11
	Monitoring/Control Period	11
3.1.2	How Globus GRAM works	11
3.2	Co-allocation Strategies	12
3.2.1	Job Request Types	13
3.2.2	Scheduling Policies	14
4	The Design of a Co-allocation Service in DAS	16
4.1	Overview of DAS	16
4.2	Problem Statement	17
4.3	The Structure of the Dynamic Co-allocation Service	18
4.3.1	General Overview of the Structure	18
4.3.2	The Resource Monitor	19
	A Linux-command-based Resource Monitor	20
	An MDS-based Resource Monitor	20
	A PBS-based Resource Monitor	20
4.3.3	The Resource Broker	22
	Fitting an Unordered Job	22
	Fitting an Ordered Job	22
4.3.4	The Scheduler and the Wait-Queue	23
4.3.5	The Co-allocator and the Run-List	24
4.4	Communication and Synchronization between Threads	26

5	Implementation of the Co-allocation Service	29
5.1	Modules and Data Structures	29
5.1.1	Globus Modules	29
5.1.2	Support Modules	30
5.1.3	Main Modules	31
	The <i>resmonitor</i> Module	32
	The <i>broker</i> Module	33
	The <i>scheduler</i> Module	33
	The <i>coalloc</i> Module	33
	The <i>dcs</i> Module	35
5.2	Additional Supportive Utilities	35
6	Experiments with the Co-allocation Service	36
6.1	The Poisson Application	36
6.2	General Conditions for the Experiments	37
6.3	Experimental Results	37
7	Conclusions and Future Work	42
A	User's Manual for DCS	43
A.1	Installing and compiling the DCS package	43
A.2	Preparing user applications and job requests	44
A.3	Submitting job requests through <i>dcs</i> command	44
A.4	Specifying which clusters are available/involved	45
A.5	Analyzing the results	45
B	An Example of Unordered Job Request	46

Chapter 1

Introduction

The advance of science and technology motivates people to do various massive and high performance computation that need enormous computational power (e.g., scientific visualization, space exploration). Most of the time, however, the required computational power cannot be offered by a single cluster of resources (i.e., a set of resources connected by a local network). Collaboration between resources in different clusters is needed. However, that is difficult to achieve since those clusters might consist of heterogeneous resources and might be managed by different local administrative domains.

People started to think how to obtain computational power through a similar public infrastructure as for electricity or water, although it is restricted by cluster boundaries and local administrative domains. This situation is similar to the case of electricity many years ago before the power grid was developed. Electrical power was there, but people do not have the same access to it so there is no much benefit obtained. This condition changed after a power grid was built to give people the opportunity to have reliable, low-cost, universal but controlled access to a standardized service of electricity.

Inspired by the power grid, experts have been developing an infrastructure that would give the same advantages to obtain computational power, which is usually called as computational grid (or simply grid). With computational grids, people will be able to access or use heterogeneous, high-end resources (application software, processors, memories, storages, etc.) without having to know about the location or structure of the resources.

Computational grids can be implemented as virtual organizations [8], which are characterized by resource sharing between different institutions. They can access directly resources they need but still in the control of the original owners, to achieve the common goal, e.g., performing scientific visualizations. Virtual organizations are dynamic/flexible, meaning that their structure (in terms of the resources involved, the nature of the access permitted, and the participants) can be modified depending on the need. Therefore, virtual organizations are commonly very complex.

A subsystem of a grid called multicluster system is simpler. It is also a wide-area network, consisting of several cluster of resources. Unlike a virtual organization, however, a multicluster system is static in that its structure cannot be modified after it has been installed in a certain way.

One of the tools to develop and maintain grids is the Globus Toolkit [12]. It provides some basic services that can be developed to implement grid functionalities. The basic services are Security Service, Data Management, Information Service, and Resource Management. Since a multicluster system is a subsystem of grid, it can also employ Globus to build similar grid functionalities.

One of the grid functionalities is *co-allocation* which is defined as allocating multiple resources in different clusters at the same time. Each cluster is usually managed by a local scheduler such as PBS [13], LSF [14], and Condor [11]. Co-allocation is very critical in a grid because by using multiple resources in different clusters at the same time, we can optimize the performance

of the whole system (e.g., response time, utilization, etc.). Nevertheless, co-allocation also has challenges inherited from the grids (e.g., heterogeneity of resources and difference in local administration domains) and a challenge of how to handle failures.

There are two aspects of co-allocation that need to be addressed:

1. a co-allocation mechanism, i.e., the interaction between functional components of the Globus resource management architecture to implement the co-allocation.
2. a co-allocation strategy, i.e., an algorithm or policy devised to handle the variety of job request types, the selection of a job from the job queue, and the selection of appropriate clusters where the available resources exist.

Co-allocation mechanisms have been implemented in the Globus Toolkit through its Resource Management components: the Dynamically-Updated Request Online Co-allocator (DUROC), and the Globus Resource Allocation Manager (GRAM). DUROC provides API functions to distribute each component of a job request to their destination clusters, while GRAM is located in each destination cluster waiting for a subjob from DUROC and interacts with a specific local scheduler.

In this thesis project, we study about the two aspects of co-allocation and use the Distributed ASCI Supercomputer (DAS) as the test-bed. It is a homogeneous multicluster system connecting five Dutch universities. Each cluster represents the local network for a participating university. Due to its homogeneity, DAS accommodates simpler ways to study co-allocation issues. DAS employs the Portable Batch System (PBS) as the local scheduler for each cluster, and Globus as the tool to build and maintain a grid environment.

Although DAS has implemented co-allocation mechanisms provided by Globus, we identify some problems due to some characteristics performed by DUROC. DUROC implements what we typically call static co-allocation, i.e., users must always specify the job requests in fixed and complete manner. For example, users must identify each destination clusters at the beginning although the clusters might be overloaded by other jobs so that they cannot receive the submitted jobs anymore. By employing some co-allocation strategies and consequently appending some mechanisms to the initial ones, we would have eliminated or at least reduced the problems. We design this proposed solution in what we call a dynamic co-allocation service which will be implemented in DAS. This co-allocation service may also be viewed as a prototype of the more general co-allocation services that can be feasibly implemented in multicluster systems.

This report serves as the documentation of the thesis project and will be structured in the following way. Chapter 2 will overview the concept of grid computing enabled by the Globus Toolkit, including the grid architecture and basic services provided by the Globus Toolkit. Chapter 3 will focus on the concept of co-allocation in grids, which are the general forms of multicluster systems. This concept includes co-allocation mechanisms, both in DUROC and in GRAM, and co-allocation strategies. So, both chapters present theoretical backgrounds for the whole thesis project. Chapter 4 concentrates on the design of a dynamic co-allocation service, as a proposed solution to the initial co-allocation mechanism in DAS. This chapter identifies problems that reside in the initial co-allocation mechanism, the structure of the proposed co-allocation service, and the communication and synchronization between threads in the proposed co-allocation service. Chapter 5 will discuss about the implementation issues regarding the co-allocation service that has already discussed in the previous chapter. This will include the structure of modules used to develop the proposed co-allocation service, and some additional utilities that are useful. Chapter 6 will discuss about some experiments that have been conducted to evaluate the proposed co-allocation service. There are also some simple analysis to the results of the experiments. The whole report will be concluded in Chapter 7, along with some ideas about future work.

Chapter 2

Grid Computing in the Globus Environment

The way of doing computation by using computational grids is usually called as grid computing. Nowadays, various companies and institutions have been developing their own ways to create and maintain computational grids. In this chapter, however, we will overview the concept of grid computing using Globus Toolkit as the environment. First, we will present an architecture of grids in terms of Globus definition, and then present the basic services provided by the Globus Toolkit.

2.1 The Grid Architecture

A computational grid is developed according to a general pattern called the grid architecture [8] which comprises some layers. Each layer consists of several components that share common characteristics, and it can build on capabilities and behaviors provided by any lower layer. The grid architecture is actually analogous to the Internet Protocol (IP) architecture, as shown in Figure 2.1.

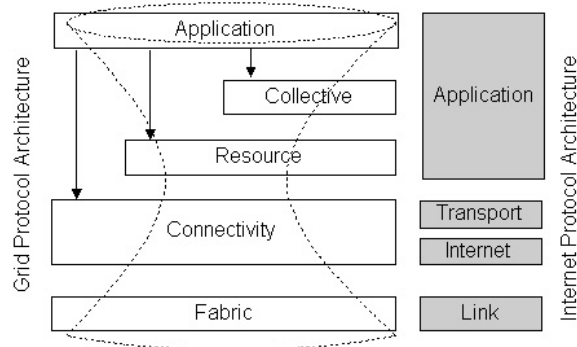


Figure 2.1: The grid architecture compared with the IP architecture.

The grid architecture employs the principles of an hourglass. The closer we go to the neck of the hourglass, the fewer protocols we meet. However, those fewer protocols are extremely important. The Connectivity and Resource layers become the neck of the hourglass. The protocols at these layers can be implemented on the top of various resource types defined at the Fabric layer and are used to construct application-specific services at the Collective layer.

Each layer applies some specific protocols according to services provided in that layer. For each service, there are API and SDK to enable users to build applications or just make use of the services.

The Fabric layer has function as the interfaces to local control. It is analogous to the Link Layer in IP architecture. It defines resource types to which shared access will be done, such as computational resources, storage systems, catalogs, network resources, etc. Resources do not only encompass physical, but also logical entity, such as distributed file system, computer cluster, etc.

The Connectivity layer makes communication easy and secure. It is analogous to the Transport and Internet Layer in IP architecture. It defines core communication and authentication protocols required for grid-specific network transaction. Communication protocols, which include transport, routing, and naming protocols, enable the exchange of data between Fabric Layer resources.

The Resource layer accommodates how to share individual resources. It corresponds to the Application layer in the IP Architecture. It defines protocols, APIs, SDKs for the secure negotiation, initiation, monitoring, control, accounting, and payment of sharing operations on individual resources. It is only concerned with individual resources and not with the issues of global state and atomic actions across distributed collections.

Unlike the Resource layer which deals with individual resources, the Collective layer has to coordinate multiple resources as a group. It is also mapped to the Application layer in the IP Architecture. It ranges from general purpose to highly application or domain specific.

The Application layer comprises user applications that operate within a grid environment. The applications are constructed in terms of services defined at any layer. At each layer, the protocols are defined to provide access to some useful services. At each layer, the APIs are also defined whose an SDK to exchange protocol messages with the appropriate service(s) to perform desired actions.

2.2 Globus Basic Services

Globus Toolkit [12] provides some basic services that should be found in a grid. Each service needs a standard protocol or component because a grid deals with diverse types of resources.

2.2.1 Security Service

This service is implemented by the Globus Security Infrastructure (GSI). Its main goal is to provide secure authentication, authorization, and communications over an open network within a grid. To implement that, GSI utilizes Secure Socket Layer (SSL) protocol, public key encryption, and X.509 certificates.

In Globus, the authentication is done through a command named *grid-proxy-init*. By that command, each user gets a temporary proxy certificate which is created based on his private key. The proxy is needed before a user submits any job or transfers any data within a grid. In some cases, depending on the configuration, it is also needed when a user queries information about resources from the grid information services.

GSI enables the so-called *single sign-on* mechanism in the authentication process. It means that once a user logs on to a grid, he can access all resources in the corresponding grid without having to be authenticated again in each resource. GSI also enables *delegation* mechanism which means a proxy can initiate another authentication if necessary in any grid environment on behalf of the proxy owner.

After the authentication process, GSI will check whether the authenticated user is authorized to do some activities in the grid. This is done by mapping the user ID contained in the proxy certificate into a local user ID in a cluster of grid resources. Every cluster has a list of authorized users, and if the mapping succeeds, the user can get access to those resources.

A user can also increase security measures by adding other mechanism like GSISsh to provide a secure shell.

2.2.2 Data Management

There are two purposes of data management in Globus: enabling data transfer and access in secure and efficient manner, and managing data replication within a grid. Data transfer and access is implemented by GridFTP and Globus Access to Secondary Storage (GASS), while data replication is implemented by some API functions such as *globus_replica_manager* and *globus_replica_catalog*.

GridFTP is actually an extension of File Transfer Protocol (FTP), the popular data transfer protocol in the Internet. To work properly, GridFTP consists of a server and a client sides. The server side is implemented by the *in.ftpd* daemon and the client side by the *globus-url-copy* command and other associated API functions.

Some features added upon the FTP to build the GridFTP are:

- GSI's authentication mechanism, such as single sign-on.
- Third-party data transfer that allows a third party to transfer files between two GridFTP servers, while the standard data transfer only allows files transferred from a GridFTP server to a GridFTP client.
- Parallel data transfer which allows multiple TCP streams be transferred to improve the aggregate bandwidth.
- Partial data transfer that enables transferring a portion of a file.
- Extended fault recovery methods to handle network failures, server outages, etc.

Transferring files between two file servers can also be done with GASS. However, unlike GridFTP which transfers data files in large sizes, GASS is used to transfer executables (associated to user jobs) as well as their input, output, and error files if necessary. Therefore, GASS is closely related to the job submission task coordinated by the Globus Resource Management (Section 2.2.4).

File transfer managed by GASS is also called file staging. The process of transferring executables and their input files from their user-executable hosts to the hosts where they will be executed (execution hosts) is called staging-in. On the other hand, the process of transferring output files from the execution hosts back to the initial user-executable hosts or the hosts where the user jobs are submitted previously (submission hosts) is called staging-out.

Like GridFTP, GASS also has its server and client sides. A GASS client will be automatically started by the Globus Toolkit in every execution host whenever a job submission occurs. However, a GASS server must be started manually only when a file staging is needed. It must be started in the same machine where the executable or its input files are stored.

The GASS servers have their own URLs so that GASS clients can identify their locations, usually with the `https://` prefix to include GSI capabilities. The transferred executable or data files will be stored in a directory called *cache storage* in the execution hosts, and will be removed from the caches after the jobs have completed their execution. If the execution results in output files, these can be staged to their submission hosts or user-executable hosts. Figure 2.2 shows how GASS protocol works in transferring files.

For most cases, submission hosts and user-executable host are the same machine. It means, the host where a grid job will be submitted will also have a local GASS server so that the executable and input files can be transferred from the same host.

2.2.3 Information Service

Information about configuration and status of every resource involved in a grid is very important to select the available and appropriate resources based on the job specification. Globus Toolkit provides Monitoring and Discovery Services (MDS) to collect the information from every grid resource and to give a uniform and portable way for users to retrieve the information. This is

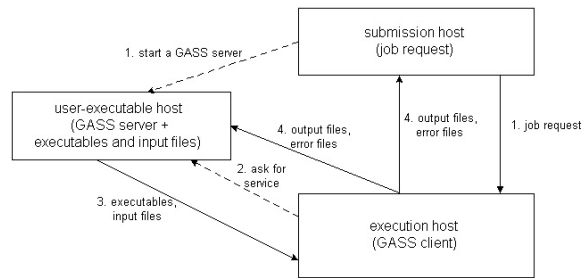


Figure 2.2: Consecutive steps when starting a GASS server in a host different from the submission host.

necessary since a grid has various types of resources, and each type has its specific configuration or status information. Moreover, each cluster/site in a grid could have its own technique to store the information from its resources. To meet that requirement, MDS is built on the Lightweight Directory Access Protocol (LDAP).

LDAP is a client-server protocol that provides directory service which is reliable and portable. The information is stored in a hierarchical structure called Directory Information Tree (DIT). Some of the rules for DIT are given in the following:

1. Each node of the DIT is called an entry.
2. An entry is a collection of attributes. Every entry is identified by a unique Distinguished Name (DN).
3. An attribute has a unique name, a type, and could have one or more values, which is represented as a tuple of (attribute-name, attribute-values).
4. An object class is a special attribute whose value defines the type of other regular attributes.

Each DIT belongs to a single LDAP server. LDAP server is implemented by a daemon process called *slapd*. Every user can retrieve the information from an LDAP server through an LDAP client which is usually implemented by *ldapsearch* command. There are also some commands that can be used to add, modify, or delete an LDAP information. Nowadays there are many ways available to access LDAP directories, such as web browsers, Java-made browsers, Linux-based browser, etc.

MDS is constructed of several components, namely, Grid Resource Information Service (GRIS), Grid Index Information Service (GIIS), Information Providers, and MDS clients.

GRIS is a directory containing information from resources in its local site/cluster. It must notify its existence by registering all the information it holds to a higher-level directory called GIIS. GRIS collects the information of its local resources through information providers. An information provider is a program responsible for converting a resource information into a data format defined in the schema or configuration files. Each information provider is specific to a particular information type. Some information types collected by GRIS through information providers are:

- Static host information (e.g., OS name and version, CPU name, speed, cache size)
- Dynamic host information (e.g., CPU load average, queue entries)
- Network information (e.g., bandwidth, latency)
- Highly dynamic information (e.g., free physical/virtual memory, number of idle processors)

Every information collected in GRIS will be cached for a configurable period of time, called time-to-live (TTL), to reduce the overhead of retrieving the required information from information providers every time a query comes. If there is no query coming within the TTL, the corresponding information will be removed from the GRIS. Then, if a query comes later, GRIS will retrieve the required information from its provider again.

A GIIS is a directory maintaining indexes of all information owned by GRIS or other GIIS instances which have registered to that particular GIIS. It means that MDS allows to have a hierarchical structure of GIIS instances. Every GIIS instance has its own identifier to enable users to access it directly if necessary. GIIS also maintains a cache and its TTL like GRIS does. Whenever a user queries an information from a GIIS instance, if the information doesn't exist in the cache, the GIIS will query GRIS or another GIIS instance which maintains that information.

2.2.4 Resource Management

The objective of the Globus resource management is job submission and control. A job is a process or set of processes that will run according to a certain specification (e.g., what application should be executed, which resources should be used, how much memories will be needed, etc). Such a job specification is also called *job request*, and this term is often used interchangeably with job. However, we should distinguish a job from an application that will be executed. In Globus environment, a job request is written in a common and portable language called the Resource Specification Language (RSL), which will be explained in more detail later on.

Job submission is accomplished by processing the job request, and allocating some particular resources in local or remote clusters based on the job request. With the help from GSI, users do not have to login their ID every time a resource is allocated.

After submitting jobs, Globus can control the jobs by doing things such as monitoring the status of the allocated resources and the progress of the jobs, canceling jobs that fail to complete, which means releasing the resources allocated for them, adding some new portions from or removing some existing portions to a job, and modifying some parts of a job content.

Globus resource management is also considered as the abstraction of resource heterogeneity from users.

Resource Specification Language

The RSL grammar is quite simple. It defines that a job request may consist of a simple relation, a single component (subjob) or several subjob. A simple relation is formed by a tuple of (param-name operator param-value). The *param-name* is simply a character string with or without quotes. The *operator* is one of the boolean logical operators, such as '<', '>', '=', etc. The *param-value* is a string of alphanumeric with or without quotes. A subjob may consist of several simple relations or several further subjobs, and they are connected together by an operator. If the operator is '&', the subjob is called to have conjunction relation. If the operator is '|', the subjob is called to have disjunction relation. Several subjobs must be connected by the '+' operator.

For detail, we can look at the following grammar in BNF.

```

specification -> request
request -> multirequest | conjunction | disjunction | relation
multirequest -> '+' request-list
conjunction -> '&' request-list
disjunction -> '|' request-list
request-list -> '(' request ')' request-list | '(' request ')'
```

```

relation -> param-name operator param-value
operator -> '=' | '>' | '<' | '>=' | '<=' | '!='
value -> ('[a..Z][0..9]['-'])+
```

The following example shows the application 'myprog' that needs 5 nodes with at least 64 MB memory, or 10 nodes with at least 32 MB.

```
& (executable = myprog)
  ( | (& (count = 5)(memory>=64))
    (& (count =10)(memory>=32))
  )
```

RSL also supports simple variable substitutions. They are made to refer variables that are not yet defined. For example,

```
&(rslSubstitution=(URLBASE "ftp://host:1234"))
(rslSubstitution=(URLDIR $(URLBASE)/dir))
(executable=$(URLDIR)/myfile)
```

is equivalent to the following statement:

```
&(executable=ftp://host:1234/dir/myfile)
```

Chapter 3

The Concept of Co-allocation in Grids

When a grid application is executed, there are occasions when a single cluster alone cannot provide all the resources needed by the application. If the application waits until there are enough resources in that cluster, it will suffer bad performance in terms of response time, for instance. Therefore, it is necessary to decompose a job associated to the application into components (subjobs), distribute them to several clusters, and run them at the same time. This effort is often referred to as *co-allocation*, which is by definition simultaneously allocating multiple resources in different clusters. Some research such as [2] and [7] has proved that co-allocation, to some extent where communication overhead does not dominate, increases the performance (e.g., response time, utilization) of a grid. Co-allocation also increases the probability for large applications to be executed.

However, grids have some inherent challenges for co-allocation to be implemented [5]. Some of them are the heterogenous properties of resources in a grid, and diversity of local administrative domains over clusters. Such properties will give more difficulties to coordinate communication between processes derived from the same job but distributed in diverse clusters. Besides that, there is a problem of how a grid responds and handles a failure in allocating resources such as competing demands by other applications for the resources or hardware failures in the resources themselves [6].

This chapter will explore what mechanism and strategies have been devised to implement co-allocation in grids.

3.1 Co-allocation Mechanism

The term *mechanism* here is concerned with the *technical procedures* of co-allocation, such as decomposing jobs, distributing subjobs, monitoring execution of those subjobs, etc., regardless any strategy given in Section 3.2. A co-allocation mechanism is specific to a certain grid infrastructure (e.g., Globus), while a co-allocation strategy is general and independent from any grid infrastructure.

Different co-allocation strategies can be implemented by the same mechanism. Vice versa, a particular strategy can be implemented by different kinds of mechanisms. In this research we use a co-allocation mechanism given by two components of the Globus Resource Management, DUROC and GRAM.

3.1.1 How Globus DUROC works

Globus DUROC holds the primary role in the co-allocation mechanism. It provides a set of functions (API) to run the mechanism, therefore it needs another entity, that is a higher-level

program called the co-allocator (co-allocation agent), to make use of the API.

The co-allocation mechanism consists of three main periods which are defined in [6] as *allocation*, *configuration*, and *monitoring/control*. Later on, we will see that Globus GRAM takes part in the co-allocation mechanism in the lower level. Figure 3.1 shows the three main periods of the co-allocation mechanism.

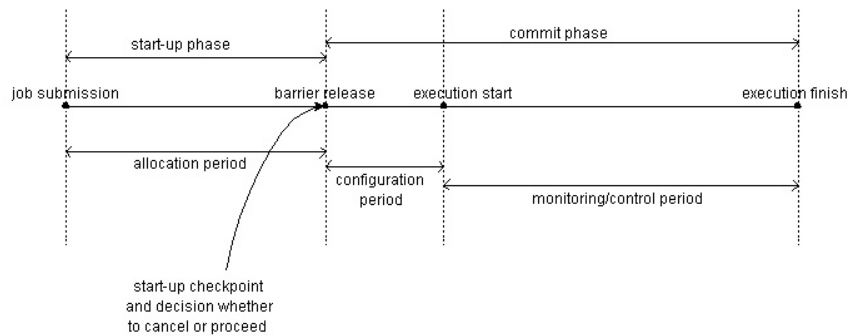


Figure 3.1: The three main periods and DUROC barrier mechanism.

Allocation Period

In the *allocation* period, the first thing a co-allocator does is to decompose a job request into components (subjobs) by using a parsing mechanism. It eliminates DUROC-specific attributes (i.e., *ResourceManagerContact*, *label*, *subjobCommsType*, *subjobStartType*) from each subjob, and add several environment variables that are useful to execute every subjob in its corresponding cluster. After that, the co-allocator, through DUROC, sends those subjobs to their destination clusters simultaneously, and guarantees the atomicity of the job start-up, i.e., all of its subjobs must be able to allocate resources they request or the entire job must be canceled.

In order to guarantee the start-up atomicity, DUROC applies a barrier mechanism, which is described in the following.

1. **Start-up Phase:** After decomposing a job request and sending each subjob to its destination cluster, the co-allocator waits until it detects a failure or receives a message from every destination cluster confirming that the subjob has entered its barrier. We must keep in mind that if a subjob can enter its barrier, it means the local resource manager (e.g., PBS) of the corresponding cluster must have been able to allocate resources for the subjob. So, the message is also confirming that the corresponding subjob has successfully gained its resources. All the subjobs will be released from their barriers if the co-allocator has received messages from all the destination clusters so that the entire job can continue to the Commit Phase. In case there is any subjob that cannot reach its barrier due to some error (e.g., resource insufficiency, hardware failures, network congestion, etc.), there will be a failure message sent to the co-allocator, and the co-allocator will assume that the subjob fails to gain its resources (start-up failure) and will act according to a transaction policy in the Commit Phase.
2. **Commit Phase:** If all the subjobs have been released from their barriers, it means the whole job has a successful start-up and can proceed to the next period of co-allocation (i.e., configuration and monitoring/control). If the co-allocator detects a start-up failure of a particular subjob, the co-allocator will act according to one of the following transaction policies.
 - **Atomic transaction:** The co-allocator will cancel the entire job.

- **Interactive transaction:** The co-allocator will not immediately cancel the job, but it looks first at the type of the resources requested by the subjob. If they are *required* resources, the entire job will be canceled. If they are *interactive* resources, they will be replaced by others or removed from the RSL script and the updated job request can proceed. The replacement of the resources is done by deleting the corresponding subjob, adding a new subjob, or just modifying the specification for resources in the troubled subjob. If they are *optional* resources, the co-allocator will ignore them and the job request can proceed without them although it might reduce its performance.

Figure 3.1 also shows the two phases of this commit protocol. The resource type of required, interactive, and optional can be specified in RSL attribute labeled *subjobStartType* with "strict-barrier", "loose-barrier", and "no-barrier" values", respectively.

DUROC barrier mechanism requires every grid application to invoke a barrier function, namely *globus_duroc_runtime_barrier()*, that causes every subjob spawned from the application enters a barrier condition. Applications using MPICH-G2 library do not have to invoke the barrier function explicitly since the function has been included already in an MPICH-G2 function called *MPI_Init()*. Likewise, to release the group barrier if all subjobs have entered their barriers, DUROC requires the co-allocator to invoke a barrier-release function called *globus_duroc_barrier_release()*.

We cannot guarantee that a job will complete its execution successfully, even though it is been started up successfully in the allocation period.

Configuration Period

Every job that has successfully gained its resources in the allocation period will move to the *configuration period*. At this point, job processes have been created by the local resource manager in all allocated resources. Each process is in a processor (CPU). Upon created, before the execution time, each newly created process needs to configure several variables, e.g.its own rank, the size (number of processes) of the whole job, the number of subjobs in the whole job, the number of processes in a specific subjob.

Those processes also need to communicate each other by sending and receiving messages. Direct communication can only be done between two processes in the same subjob or between two processes whose rank '0' in different subjobs. Therefore, if a process wants to communicate with another process in different subjobs must use the 'rank 0' processes in their own subjobs as intermediaries.

Monitoring/Control Period

After configuration period, all job processes over clusters start to execute the application code. The whole job will not complete unless all its processes complete their execution. In the mean time, we can monitor and control the execution of those processes as a collective unit, rather than individual processes.

As an example, DUROC provides an API function *globus_duroc_control_subjob_states()* for monitoring states change of every subjob, and function *globus_duroc_job_cancel()* for canceling the whole job which is an example of control.

3.1.2 How Globus GRAM works

The component of the Globus Toolkit to interact with the local scheduler (PBS in this case) in a single cluster is GRAM server. It is composed of a gatekeeper, a job manager, and a reporter, which can be seen in the Figure 3.2.

The GRAM gatekeeper is a daemon process situated on every cluster. It works similar to the *inetd* daemon but it utilizes GSI to authenticate the user. It keeps listening whether there is a job request or client command coming. If so, it receives a job request (in RSL) submitted by a

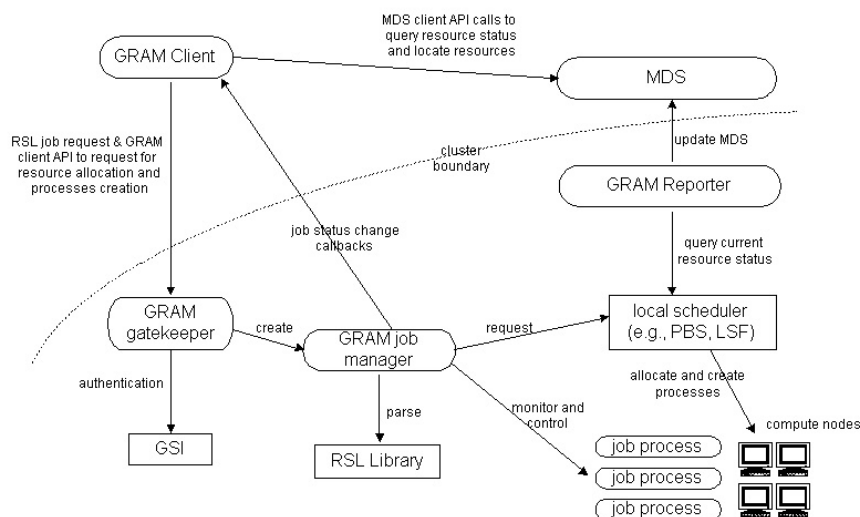


Figure 3.2: Interface between GRAM and a local scheduler in a single cluster

GRAM client. The job request must be sufficiently concrete such that GRAM should not consult again with the client. The gatekeeper also receives the user information brought in the client API. As soon as the gatekeeper receives the subjob request and user information, it performs mutual authentication between user and resource, so that the user knows the resource he requires and resource knows the user who will allocate it. After that, the gatekeeper determines a local user name that will become the representative of user in each local host where the job will be executed.

After authentication, the gatekeeper starts a job manager by *fork* or *su* command. Job manager is a process activated in the same cluster to directly interact with the local resource manager. The gatekeeper can leave the job manager to communicate with the client (e.g., informing the status of the actual job) so that the gatekeeper becomes ready to receive the next job request. Upon started, the GRAM job manager will parse the subjob request, i.e., the GRAM-specific attributes, with the help from the RSL library. The result of the parsing will be passed on as a request to PBS, which will allocate the appropriate nodes/resources and create the actual processes running on those resources. During the time when the processes is running, the job manager must keep monitoring and controlling (e.g., canceling) them, and inform the changes of job state to the client. Job manager lives for a temporary moment, while the gatekeeper lives is permanent. It is created just for a particular job request and will terminate if the job for which it is created has terminated. If there are several job requests received by the gatekeeper, there will be several job managers; one for each job.

The GRAM job manager will transform the RSL file which contains the job request into a PBS job script. Then it will call a PBS command *qsub* to submit the subjob request to PBS. The job manager can also use other PBS commands to do corresponding functions, such as *qstat* to monitor the status of processes created after the subjob, and *qdel* to kill those processes when it is needed.

The GRAM reporter has the role to query job status from PBS, and update the information in the MDS. It is supposed that the reporter does its work regularly to the MDS, but in fact, it doesn't occur quite often.

3.2 Co-allocation Strategies

The term *strategy* means how we manage and organize the placement of jobs to the suitable resources to achieve the co-allocation goal. First, we discuss about some job request types and

how to fit a job of each type to the available resources. Then, we also look at the variety of scheduling structures in case there are multiple jobs submitted at the same time. Co-allocation strategies are usually independent from any specific framework.

3.2.1 Job Request Types

A job request that needs co-allocation consists of multiple components, namely subjob requests. Each subjob request is associated with a single cluster; thereby sometimes such a job is also called as multi-cluster job.

In [3], Anca Bucur classifies a job request into one of the following types:

- **Ordered:** Each subjob specifies the number of processors it requires and identify the cluster from which those processors will be allocated.
- **Unordered:** Each subjob specifies the number of processors it requires but *does not* identify the cluster from which those processors will be allocated.
- **Flexible:** This job specifies the *total* number of processors required collectively from multiple clusters, instead of the number of processors for each subjob.
- **Total:** This job specifies the total number of processors required from *a large single* cluster, instead of multi clusters. By this nature, a job of this type is also called a single-cluster job.

In order to fit a job request to the available resources in multiple clusters, we need a job-fitting algorithms which depends on the job request type [3].

1. For an **ordered** or **total** job, each subjob can be fit straightforwardly to its associated cluster, and it will be obvious whether or not the job request fits.
2. For an **unordered** job, subjobs are sorted according to their sizes in descending order, and one of the following methods must be applied for each subjob from the beginning to the end of the order.
 - **First-Fit:** select the first cluster that fits the subjob.
 - **Best-Fit:** select the cluster with the smallest number of idle processors on which the subjob fits.
 - **Worst-Fit:** select the cluster with the largest number of idle processors on which the subjob fits.
3. For a **flexible** job, subjobs will be distributed across all the available clusters according to one of the following methods.
 - **Cluster Filling:** Clusters are sorted according to their load (i.e., the number of busy processors). Started from the least-loaded, each cluster will be loaded completely (i.e., all its processors get allocated) until the total number of the required processors have been met. As the result, there will be potential load imbalance in that some clusters still have idle processors while some others have been fully loaded. This method is devised to minimize the number of clusters allocated to execute the job, thereby minimizing the overhead of inter-cluster communication.
 - **Load Balancing:** Clusters are loaded in such a way to balance the load among them. Consequently, there is possibility that all clusters still have idle processors, but all of them have the same load. Therefore, the utilization of the whole system can be increased.

A research conducted by Ernemann et al. [7] discusses about the benefit of using grids to run parallel jobs. They define three possible strategies to run parallel jobs which are known as local job processing, job sharing, and multi-site computing. With local job processing, a job can only be executed on processors in its local cluster. In job sharing, a job can be executed in another cluster, but it must be a single cluster rather than multiple clusters. Only in multi-site computing, a job can be executed in multiple clusters.

Multi-site computing strategy basically use the same principle as job-fitting algorithms for flexible jobs (i.e., cluster filling and load balancing). The research shows that multi-site computing performs the best among the strategies. Again, what Ernemann et al. have done actually proves the importance of co-allocation.

3.2.2 Scheduling Policies

It is possible that multiple job requests are submitted to a grid at relatively the same time. It means there will be some jobs running in the grid while others must wait in a job queue to be processed. Therefore, a grid often needs a scheduler to manage multiple jobs in the queue. Such a scheduler is also called meta-scheduler [10]. It makes decisions of when a job can be processed and which job to be selected for that. The selected job will be run in one or more clusters simultaneously according to the job specification.

The following are some scheduling policies that can be used to select a job request from the queue [2]:

- **First Come First Served (FCFS):** The first incoming job request (in arrival) get the highest priority. In every re-scheduling action, the first job is checked whether it fits the available resources. If so, it will be executed immediately. If not, the job must wait until the next re-scheduling action takes place.
- **Fit Processor First Served (FPFS):** In every re-scheduling action, the scheduler gives the highest priority to a first job that can fit the available resources, not necessarily the first job in arrival. If there is such a job, it will be executed immediately. If all jobs cannot fit, they must wait until the next re-scheduling action takes place.
- **Backfilling:** This policy is similar to FPFS, but in every re-scheduling action, the execution of the next job must not be delayed by the smaller job (i.e., the job which requires fewer processors) that jumped over it. The job that jumps over the first job is also called the "backfilled" job. In fact, there are two versions of backfilling: EASY and conservative. The EASY version has to make sure that the execution of the first job will not be delayed by the backfilled job, while the conservative version has to make sure that not only the execution of the first job, but also the execution of every job that has arrived prior to the backfilled job must not be delayed. For this condition, the scheduler must estimate the execution time of both running and queued jobs.

In [10], Hamscher et al. discuss various scheduling structures that could occur in computational grids. The scheduling structures are important because they are different than those of a single parallel machine. Scheduling in a single parallel machine does not consider about system boundaries.

The grid scheduling structures they define are summarized in the following.

- **Centralized:** There is only one global scheduler that will manage all jobs. Even if a job should be run in clusters other than the local cluster where it is submitted, the job must be submitted to the global scheduler. Afterwards, the scheduler will distribute the job to the suitable clusters. This structure will suffer bottle-neck condition as its disadvantage. However, this structure can provide a very efficient scheduling mechanism because the global scheduler knows all information of the available resources in the system.

- **Hierarchical:** Each cluster has its own local scheduler, but the whole system has another higher-level scheduler, which is called meta-scheduler. Like the centralized scheduling, all jobs must be submitted through this meta-scheduler before they are distributed to the suitable local schedulers. Bottle-neck condition is still possible, but as an advantage, this structure allows every cluster to have its own scheduling policy.
- **Decentralized:** This structure does not have a single global scheduler. Each cluster has its own local scheduler, and two local schedulers will communicate each other when one needs to send/receive a job or status information to/from the other. For example, if a job cannot be fit in resources of the local cluster but can be fit in the other clusters, then the local scheduler which belongs to that job will send the job to the suitable cluster. That is why the co-allocation is more difficult to implement here because communication between local schedulers produces additional overhead that would not exist in the centralized and hierarchical structures. Bottle-neck condition can be prevented, but the absence of global scheduler will result in less optimal scheduling mechanism compared with the centralized structure.

Chapter 4

The Design of a Co-allocation Service in DAS

This chapter will focus on the design of a co-allocation service which will be implemented in DAS. Therefore, we will overview the DAS first and its initial situation with regard to the co-allocation before continuing to the structure of the service itself. The discussion about the structure will encompass general overview of the whole structure, mechanism within each component, and relations between the components.

4.1 Overview of DAS

The Distributed ASCI Supercomputer (DAS) is a wide-area computer system comprising multiple clusters designed by the Advanced School for Computing and Imaging (ASCI). The DAS, which is now in its second generation is built out of five clusters, each of which is located on one of the following Dutch universities: Vrije Universiteit (72 nodes), Leiden University (32 nodes), University of Amsterdam (32 nodes), Delft University of Technology (32 nodes), and University of Utrecht (32 nodes). Its purpose is to accommodate the research of parallel and distributed computing in the participating universities.

The clusters are interconnected by SurfNet, the Dutch University backbone for wide-area communication with 100Mbit/s bandwidth. Nodes within a local cluster are connected by Myrinet-2000 LAN with 1200Mbit/s bandwidth. Inter-cluster communication is much slower than the intra-cluster (inter-nodes) communication.

DAS-2 is homogeneous in that each node has the same specification as follows:

- Two 1-GHz Pentium IIIs
- 1 GB RAM except 1.5 GB for Leiden U. and UvA, 2 GB for Vrije U.
- A 20 GByte local IDE disk (80 GB for Leiden U. and UvA)
- A Myrinet interface card
- A fast ethernet interface (on-board)
- Running on RedHat Linux OS

Each DAS cluster has one file/compile server and many compute nodes. The file servers are called fsX (X is 0,1,2,3 or 4 according to the cluster), and the compute nodes are called nodeX[0-9][0-9]. The file server is meant for developing (writing and compiling) programs, while the compute nodes are locations where the executables run. Every cluster has the same local scheduler, namely the Portable Batch System (PBS).

The homogeneity of DAS eliminates the complexity of dealing with different resource types, administrative domains, etc. This is advantageous for we can focus on the co-allocation mechanism itself.

4.2 Problem Statement

The grid environment has already been installed on the DAS with the Globus Toolkit v.2.2.4. Users can run their jobs through some client programs such as *globusrun* and *globus-job-run*. The client programs cannot submit multiple jobs at once.

Likewise, users must submit a completely specified RSL (i.e., ordered) job request to Globus because DUROC can only do its work if everything has been clearly specified in the RSL specification (e.g., the destination clusters, the number of processors in each cluster, etc). In other words, the DAS implements what we call a static co-allocation mechanism.

The fact that the DAS is used by many users may cause a user job to fail because the resources they require have been allocated to jobs of other users. If this happens, the Globus DUROC cannot do anything except immediately sending an error message telling the submission cannot be accomplished or just waiting until resources are available again. Globus DUROC does not have a time-out mechanism that can be used to mark jobs that are too long suspended in PBS queues. Also, it cannot allocate the resources in such a way that it anticipates changes of the resource state.

This situation motivates us to design a service that can automatically detect the states of clusters and dynamically allocate resources according to the current state, so the failure of a job execution can be avoided as much as possible. It also gives the users a more flexible way to specify a user job in that users do not have to specify everything in detail. To make it simpler to refer, we call this service as the Dynamic Co-allocation Service (DCS).

The DCS will have the following capabilities:

1. Users can input multiple job requests at the same time, each of which is specified in an RSL file. The system will schedule and submit the jobs to the Globus DUROC.
2. The service enables users to submit ordered and unordered jobs.
3. Users can specify either FCFS or FPFS as the scheduling policy.
4. Users can see the progress of their submitted jobs.
5. The service will do its best to successfully run every job submitted through it. It will repeatedly attempt to start a job until it succeeds or until a certain number of times is passed.
6. The system will record the progress of all jobs to a log file when they are finished.

However, the service also have some constraints:

1. Jobs submitted by users must be rigid jobs, i.e., the number of processors cannot be modified during execution.
2. The executables of user jobs must use the MPICH-G2 library to enable the co-allocation capabilities of DUROC.
3. Nested components (job components inside a job component) are not allowed.

4.3 The Structure of the Dynamic Co-allocation Service

The basic idea underlying our design is adding several higher-level components on the top of the Globus DUROC to allocate the required resources. We will overview the main components of the service and how a user job flows from the time it is submitted by a user until it completes its execution. Then, we will focus on each component in more detail. To some extent, this service resembles the so-called meta-schedulers which decide when a user job will be sent to local schedulers which are located in the destination clusters.

4.3.1 General Overview of the Structure

There are several main components of our co-allocation service, namely, the Resource Monitor, the Resource Broker, the Scheduler, the Co-allocator, as well as a waiting-job queue and a running-job list. For simplicity, we will call the waiting-job queue and the running-job list as the wait-queue and the run-list, respectively. Figure 4.1 depicts all the components, and the information flow among them.

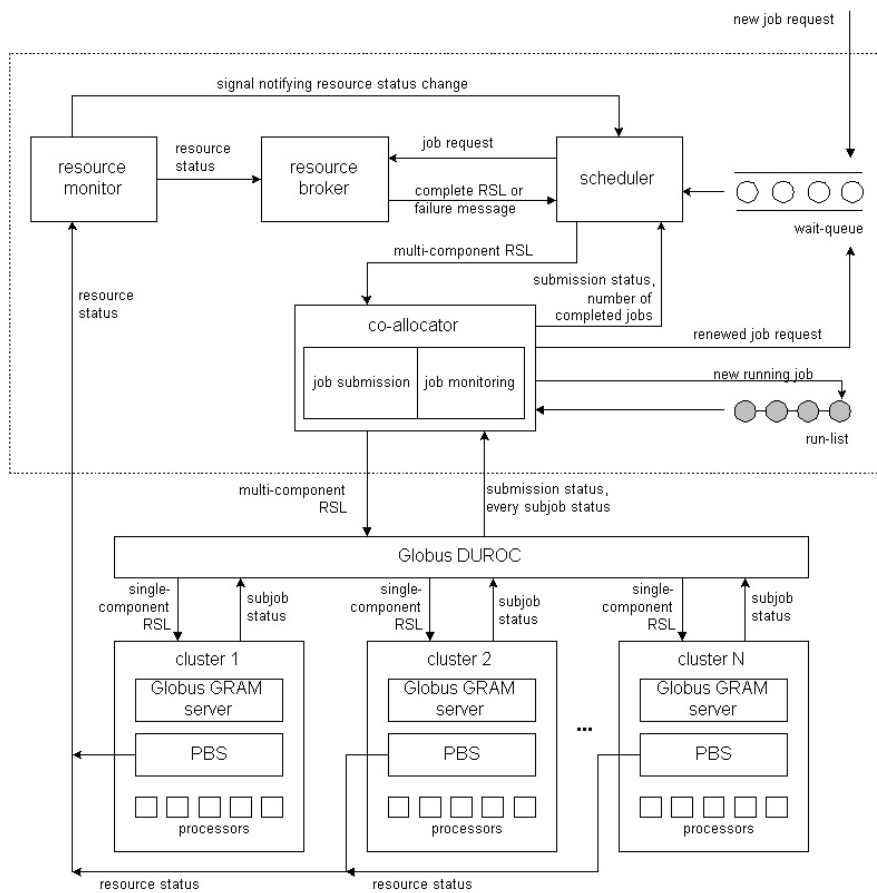


Figure 4.1: The architecture of the dynamic co-allocation service

When a user puts a set of job requests into the system, the job requests and its associated information will be inserted to the wait-queue. While this insert process is occurring, the Scheduler starts to run the (re)scheduling activity once the queue has received a job request. The (re)scheduling selects the job with the highest priority in the queue based on a scheduling policy. In the (re)scheduling activity, the Scheduler invokes the Resource Broker to make sure that the selected job request is approved to allocate resources.

When invoked, the Resource Broker receives a job request, and attempts to fit the job request with the available resources. If there are sufficient resources matching the specification, the Resource Broker approves the job request and sends the job request back to the Scheduler. If there are not sufficient resources, the Resource Broker will send a message to the Scheduler telling about its failure to find resources for the job.

In order to fit a job to resources, the Resource Broker needs to know about the current resource status, and it retrieves the information from the Resource Monitor. The Resource Monitor collects the resource status directly from PBS in each cluster. In this co-allocation service, we restrict the resources only to be the processors for simplicity.

Having received either something from the Resource Broker, the Scheduler will proceed. If the Scheduler gets a failure message from the Resource Broker which indicates that no sufficient resources are available, it will wait until there is a change of resource availability and then run the (re)scheduling again.

If the Scheduler gets a complete RSL string from the Resource Broker which indicates its approval, the scheduler will send the job request to the Co-allocator. The Co-allocator then sends the job request to the Globus DUROC, and the Globus DUROC will use its co-allocation mechanism to submit all subjobs to their destination clusters.

The success and failure of a job submission are reported by the Co-allocator to the Scheduler in a so-called submission status. If the job submission is successful (i.e., actual job processes created in the allocated processors), the Co-allocator takes the job request and puts it into the run-list, which is the place for all running jobs so that the co-allocator can monitor their progress until their completion.

However, even though the Resource Broker has approved a job request, it is possible that the job submission fails for any reason. For instance, there is a change in the resource availability while the Resource Broker is working to fit the job to the resources so that the actual job processes cannot be created in allocated resources, the executable file cannot be found etc. If this happens, the Co-allocator will cancel the job submission and tell the Scheduler about the failure. Then, the Scheduler will move the job request to the tail of the wait-queue, and run the next (re)scheduling activity.

There is another status noticed by the Co-allocator, namely the completion status. It indicates whether or not a running job can complete its execution successfully. If a running job fails to complete its execution, the Co-allocator will put the original job request back to the wait-queue so that the job request can be scheduled. If a running job can finish its execution successfully, the Co-allocator will remove the job request from the run-list because it is no longer needed. The Co-allocator does not have to report the completion status to the Scheduler. Only the the number of completed jobs, either with completion success or failure, must it report.

Now, we will see in more detail how each main component is designed.

4.3.2 The Resource Monitor

The Resource Monitor is responsible for collecting information about the resource status and for providing this information to the Resource Broker whenever required. It can retrieve various types of status information from each cluster, such as processor availability, memory availability, network load, etc. In this co-allocation service, however, we restrict the resource status only to be the processor availability for simplicity.

There are several options that can be used by the Resource Monitor to retrieve the processor availability. We will discuss each option below and which option we select to implement the Resource Monitor.

A Linux-command-based Resource Monitor

First, we look at the Linux command 'uptime' that gives the so-called *system load average*, which is the average number of job processes running in that system for a certain period of time (e.g., in the past 1, 5, and 15 minutes). We can also get the same output from the 'loadavg' file in the /proc directory. For example, if we run the command on the fs3 server, we will get the output as follows.

```
10:35am up 203 days, 23:49, 7 users, load average: 0.28, 0.14, 0.04
```

It shows a load average of 0.28 in the past 1 minute, 0.14 in the past 5 minutes, and 0.04 in the past 15 minutes. However, we do not know in which processors those processes were situated or how many processors were used to execute those processes. So basically, we still do not have the information we need.

An MDS-based Resource Monitor

A Resource Monitor can retrieve the processor availability by running the 'grid-info-search' command. This command actually invokes 'ldapsearch' which returns an LDAP information tree. An example node in the tree showing the number of idle processors of fs2 cluster can be seen in the following.

```
dn: Mds-Job-Queue-name=dque, Mds-Software-deployment=jobmanager-pbs,
    Mds-Host-hn=fs2.das2.nikhef.nl, Mds-Vo-name=local, o=grid
objectClass: Mds
objectClass: MdsSoftware
objectClass: MdsJobQueue
objectClass: MdsComputerTotal
objectClass: MdsComputerTotalFree
objectClass: MdsGramJobQueue
Mds-Job-Queue-name: dque
Mds-Computer-Total-nodeCount: 64
Mds-Computer-Total-Free-nodeCount: 42
Mds-Memory-Ram-Total-sizeMB: 0
```

We can extract the *Mds-Computer-Total-Free-nodeCount* attribute value from that output and apply the same principle to other clusters, and then we get what we need. Unfortunately, the MDS information is not quite up-to-date since the GRAM reporter is not activated all the time to collect the resource status and report it to the MDS.

A PBS-based Resource Monitor

Using PBS, this component will run 'qstat' command to retrieve the status information. The next example will show what status information are provided by the command.

```
[jsinaga@fs0 jsinaga]$ qstat -an
```

fs0.das2.cs.vu.nl:

Job ID	Username	Queue	Jobname	SessID	NDS	TSK	Req'd Memory	Req'd Time	Elap S	Time
230584	fs0.das2.versto	dque	prun job	1339	1	--	--	27777	R	335:3
node020/1+node020/0										
234975	fs0.das2.vdwijst	chemc	aNH2tNH2_p	3749	8	--	--	100:0	R	26:19
chem053/0+chem052/0+chem051/0+chem050/0+chem049/0+chem048/0+chem047/0+chem044/0										
235617	fs0.das2.vstralen	chemc	H2Po2.inp_	6225	4	--	--	48:00	R	03:11
chem077/0+chem076/0+chem075/0+chem074/0										
235665	fs0.das2.pbento	chemc	sih3c1_af	29269	3	--	--	03:00	R	00:52
chem072/0+chem071/0+chem070/0										

```

235685.fs0.das2 vdwijst chemc gOc=_pair 15166 4 -- -- 24:00 R 00:16
chem069/0+chem068/0+chem067/0+chem066/0
235693.fs0.das2 vdwijst chemc gOc_3=_pai 30600 4 -- -- 48:00 R 00:06
chem065/0+chem064/0+chem063/0+chem062/0
235698.fs0.das2 vdwijst chemc gO=_base. 28828 2 -- -- 12:00 R 00:01
chem061/0+chem060/0
235699.fs0.das2 istaicu dque prun job 17750 9 -- -- 00:16 R 00:01
node065/1+node065/0+node064/1+node064/0+node063/1+node063/0+node062/1
+node062/0+node061/1+node061/0+node060/1+node060/0+node059/1+node059/0
+node058/1+node058/0+node057/1+node057/0
235701.fs0.das2 pbento chemc cl-_sih3cl 2173 3 -- -- 02:00 R 00:00
chem059/0+chem058/0+chem057/0

```

```

[jsinaga@fs0 jsinaga]$ qstat -Bf
Server: fs0.das2.cs.vu.nl
server state = Idle
scheduling = False
total jobs = 9
state count = Transit:0 Queued:0 Held:0 Waiting:0 Running:9 Exiting:0
default queue = dque
log events = 127
mail from = adm
query other jobs = True
resources default.walltime = 00:10:00
resources assigned.nodect = 38
scheduler iteration = 60
node pack = False
pbs version = OpenPBS 2.3

```

From the example, we can get much information, such as the number of jobs running in the cluster, the number of compute nodes that are used by each job and the total number of them, the identifier of each processors executing the job processes, etc. Nevertheless, we must be careful defining the number of busy processors. The number of busy processors is not necessarily the same as the number of assigned nodes because in DAS, every node comprises two processors. Therefore, if the number of assigned nodes is represented by N , the number of busy processors must be in the range of N and $2N$.

In one case, the two processors of a compute node are assigned to execute the same job. For example, job **230584.fs0.das2** has processes running in processors **node020/0** and **node020/1** which are in the same node (**020**).

In other case, only one processor in a compute node is assigned for the same job. For example, job **235617.fs0.das2** has processes running in the processors **chem077/0**, **chem076/0**, **chem075/0**, and **chem074/0**.

There is also another situation where a processor is assigned for several job processes from different queues. For example, processor **057/0** is used to execute job **235699.fs0.das2** and **235701.fs0.das2** whose processes released from queue **dque** and **chemc**, respectively.

Therefore, the only way to get the true number of busy processors is by tracing the processors that are assigned from the first job until the last one in the 'qstat' output, and eliminating the processors that have been used more than once to avoid redundancy. So for the example above, the number of busy processors is 39, instead of 38 (which is the number of assigned nodes). Hence, the number of idle processors is 105, for the total number of processors in fs0 is 144.

This approach seems to be the most accurate among the options to implement the Resource Monitor. The Resource Monitor will do the traversal procedure explained above to every cluster in DAS-2 and store the whole information in a data structure called idle-list. The elements of an idle-list are in arbitrary order, and each of them represents a cluster with the information of the cluster ID and the number of idle-processors in that cluster. The idle-list will be read by the Resource Broker when the Resource Broker invokes the Resource Monitor.

4.3.3 The Resource Broker

The Resource Broker uses a job-fitting algorithm to fit the job request to the appropriate clusters. Referring to Section 3.2.1, it depends on the job request type to select which job-fitting algorithm to be used. First, we will see how to fit an unordered job by using the modified worst-fit algorithm. Then, we can use the same data structure to fit an ordered job.

Fitting an Unordered Job

The first thing the Resource Broker will do after receiving an unordered job is extracting the number of processors for each subjob. While the Resource Broker reads those items, it builds a linked list called the request-list. Each of its elements represents a subjob with the information of the number of requested processors and a pointer to an idle-list element. The elements of request-list must be sorted in descending order according to the number of requested processors. After that, the Resource Broker reads the numbers of idle processors provided by the Resource Monitor in the form of an idle-list. Although the idle-list is produced by the Resource Monitor, but the Resource Broker has access to modify fields in the elements of the idle-list.

After the two lists are built, the Resource Broker will traverse elements in the request-list, and try to select an idle-list element which can fit the requested number of processors. The idea here is trying to use different clusters as many as possible, and if all clusters no longer fit, try to allocate again the previously allocated clusters. If no clusters can fit the job request, then the job cannot be approved to be submitted.

To implement the idea, every idle-list element which has already been allocated for a subjob is marked. Initially, no element of the idle-list is marked. For each request-list element, the Resource Broker will do the following steps:

1. If there are still unmarked element in the idle-list, find the one with the maximum number of idle processors. If the number of idle processors is sufficient, mark the element, and subtract the number of requested processors from the number of idle processors. Otherwise, try the next step.
2. Visit every marked element of the idle-list to find the one with the maximum number of idle processors. If the Resource Broker can find the element, it will simply subtract the number of requested processors from the number of idle processors. If not, then the Resource Broker fails to fit the job request and must stop the traversal.

Every time the Resource Broker succeeds to fit a request-list element, it will switch to the next one, and start from the first step again. However, before it starts, it must check whether all the idle-list elements have been marked. If so, it must remove all the marks first.

Figure 4.2 shows an example of consecutive steps to fit an unordered job which requests 10 processors for subjob-1, 20 processors for subjob-2, and 15 processors for subjob-3.

The steps of fitting an unordered job with the modified best-fit algorithm is basically the same as that with the modified worst-fit algorithm. However, in terms of the modified best-fit algorithm, the Resource Broker must find the idle-list element with the minimum, instead of the maximum, number of idle processors.

Fitting an Ordered Job

The Resource Broker will be much more straightforward in fitting an ordered job, because both the destination clusters and numbers of requested processors are known. After building the request-list and obtaining the idle-list from the Resource Monitor in the same way as explained before, the Resource Broker will also traverse the request-list. For each request-list element, the Resource Broker will simply compare a number of requested processors with the number of idle processors associated to its destination cluster. If the number of idle processors is larger than or equal to the

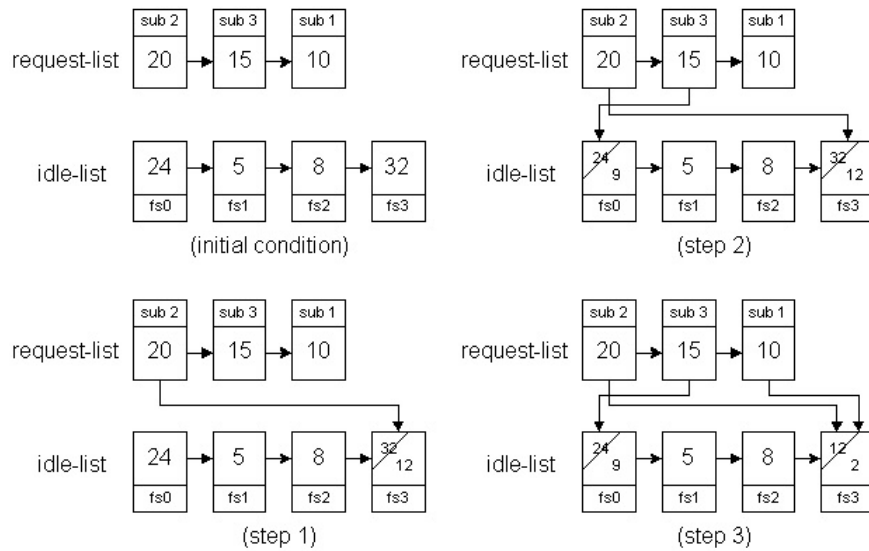


Figure 4.2: Consecutive steps of fitting an unordered job request with the modified worst-fit algorithm

number of requested processors, then the Resource Broker can fit the ordered job. Otherwise, the Resource Broker will send a message that it fails to fit the job request. However, for the idle-list element that has been compared more than once, its number of idle processors must be updated by subtracting the number of requested processors from it.

4.3.4 The Scheduler and the Wait-Queue

The Scheduler is the central component of this co-allocation service. It requires the so-called wait-queue as the place for job requests before they are submitted successfully. Those job requests are represented in separate elements, each of which keeps some information such as the following:

- the type of the job request (i.e., ordered and unordered).
- the original form of the job request which is possibly lacking the *resourceManagerContact* parameter.
- the complete form (i.e., RSL string) of the job request which is the outcome of the Resource Broker's approval.
- the amount of submission failures this job has suffered.
- the amount of completion failures this job has suffered.

The queue elements are not created or inserted by the Scheduler itself. This will be explained in Section 5.1.3 about the implementation issues. What the Scheduler will do first is waiting until the queue is filled with at least an element. As long as the wait-queue is not empty, the Scheduler will run the (re)scheduling and element-processing activities. The (re)scheduling is based on a policy (i.e., FCFS or FPFS in this case) and is intended to find the most-prioritized job request which is approved to fit the available resources. The approval is given by the Resource Broker so the Scheduler needs to invoke it during the (re)scheduling activity. For FCFS, the Resource Broker is invoked only once to examine the first job request in arrival, while for FPFS, the Resource Broker might be invoked several times, each of which is for a job request, before it finds the job which can fit the available resources. Nevertheless, for both FCFS and FPFS, the Resource Monitor will always be invoked once in a single (re)scheduling activity.

If no job request gets approval to fit the available resources, namely the (re)scheduling failure, the Scheduler will simply wait for the Resource Monitor to notify it when there is a change in resource availability. Upon receiving notification from the Resource Monitor, the Scheduler runs the next (re)scheduling activity.

If the (re)scheduling succeeds, i.e., a job request gets the highest priority and approval to fit the available resources, the information of RSL string in the representing wait-queue element will be assigned by the Resource Broker and the Scheduler sends the job request in the RSL string to the Co-allocator. The Scheduler will wait until the Co-allocator notifies it whether or not the job has been successfully submitted.

If the Co-allocator fails to submit the job request, the Scheduler increments the amount of submission failures associated to the job. If the amount is still less than a configurable maximum number, the job request will be moved from its current position to the tail of the wait-queue, and the Scheduler runs the next (re)scheduling activity. If it has exceeded the maximum number, the Scheduler will remove the associated element from the wait-queue.

However, if the Co-allocator succeeds to submit the job request, it will take the associated queue element to create a new element for the run-list. The discussion of the Co-allocator and the run-list will be given in Section 4.3.5.

Once a job request has been successfully submitted by the Co-allocator, the Scheduler will not care about whether the job can complete successfully, too. However, the Scheduler records the number of completed jobs reported by the Co-allocator, either with completion success or failure. It will compare the number with the latest number of successfully submitted jobs. As long as the number of completed jobs is still less than the number of the successfully submitted jobs, the Scheduler cannot be terminated, because it could happen that the job with completion failure will be inserted again to the wait-queue.

The whole activity of the Scheduler will be implemented as a single thread and can be summarized in the state diagram in Figure 4.3.

4.3.5 The Co-allocator and the Run-List

The Co-allocator is responsible for the job submission and job monitoring by making use of Globus DUROC API functions. Therefore, it needs the so-called run-list which stores elements representing the running jobs. Each element of the run-list keeps the following information:

- the whole content of a corresponding wait-queue element.
- the ID given by DUROC during its execution.
- a set of states describing the status of every subjob in the job.

The Co-allocator activity for job submission starts in waiting for a wait-queue element sent from the Scheduler. Once it receives an element, the Co-allocator calls the *globus_duroc_control_job_request()* function to submit the job request contained in the element to the Globus DUROC. DUROC will handle the way to split the job into multiple subjobs and distribute them to their destination clusters. The function is synchronous (blocking) so the Co-allocator must wait until the function returns. When the function returns, the Co-allocator gets the information of whether each subjob has been able to get to its destination cluster. If there is any subjob which fails to get to its cluster, the Co-allocator will call the *globus_duroc_control_job_cancel()* function to remove all subjobs associated to the job from their clusters. After that, the co-allocator will go back to waiting for another wait-queue element from the scheduler.

If all subjobs can get to their destination clusters, the Co-allocator must guarantee the job submission success by calling the *globus_duroc_control_barrier_release()* function. The function will hold until all subjobs have advanced to their own barriers. It may happen that there is a subjob that failing to enter the barrier, and that the function returns a failure message to the

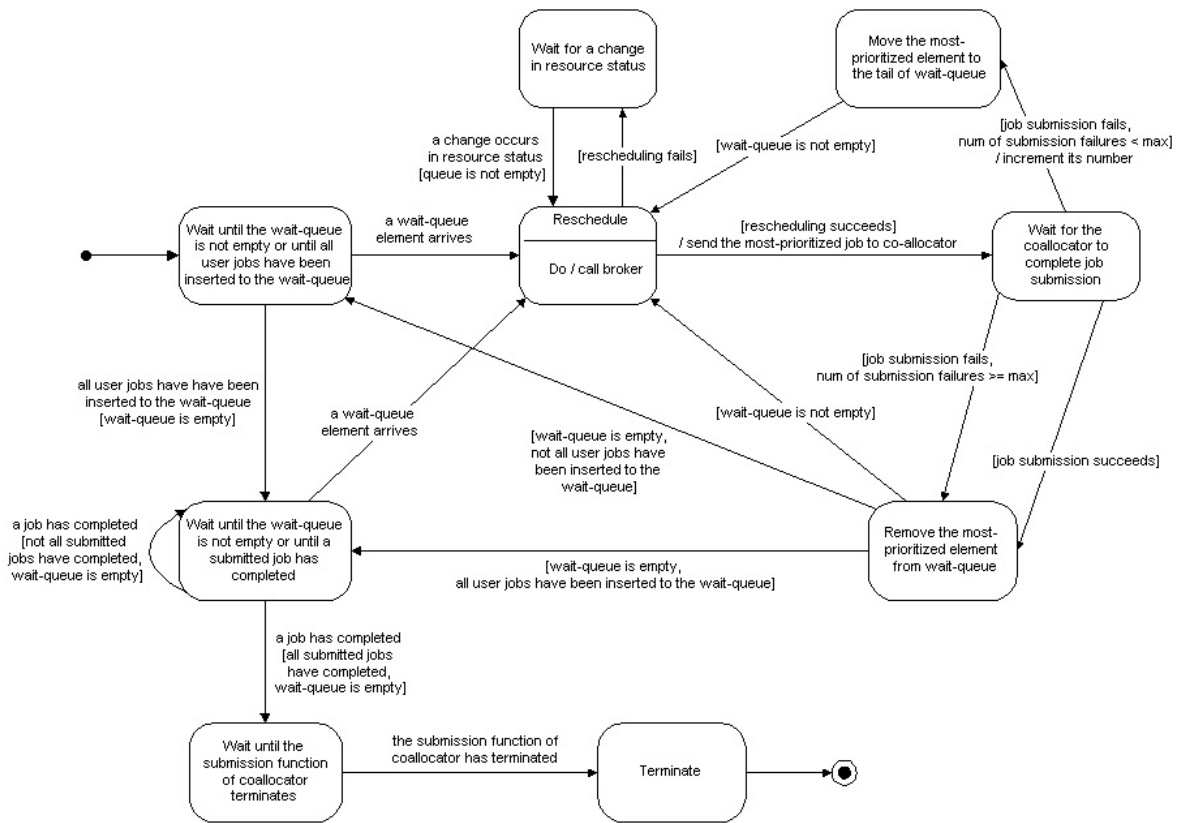


Figure 4.3: The state diagram of the Scheduler's activities.

Co-allocator. However, if the function returns correctly, all the subjobs have been released from their barriers, and the Co-allocator will notice that as a submission success and sends a success message to the Scheduler.

If the job submission succeeds, the Co-allocator takes the content of the wait-queue element and combine it with other information to create a run-list element for the job, and initializes the information with proper values. Then, the co-allocator puts the new element into the run-list.

The whole activity of the Co-allocator for job submission will be implemented as a thread, and can be summarized in the state diagram in Figure 4.4.

The monitoring activity of the Co-allocator begins when there is at least one element in the run-list. For each element in this list, the Co-allocator will call the *globus_duroc_control_subjob_states()* function to poll the status of a running job. The polling is done for every job in the run-list continuously as long as the run-list is not empty. All the status of a job will be recorded from the time its corresponding element arrives in the run-list until the job completes.

Every time the Co-allocator polls the status of a job, it checks out whether the job completes with success or failure. If all subjobs in the job have completed with success, the Co-allocator will remove the associated run-list element. If there is any subjob that completes with failure, the Co-allocator notice it as a completion failure for the job, and see whether the number of completion failures of the job has already exceeded a configurable maximum number. If so, then the run-list element associated to the job is also removed. If not, then the number of completion failures of the job is incremented, and the job is inserted to the wait-queue to be scheduled again. When a job completes either with success or failure, the Co-allocator will increment the number of completed jobs and report it to the Scheduler.

If there is any subjob that has not completed its execution, the Co-allocator will just switch to the next element to monitor its status.

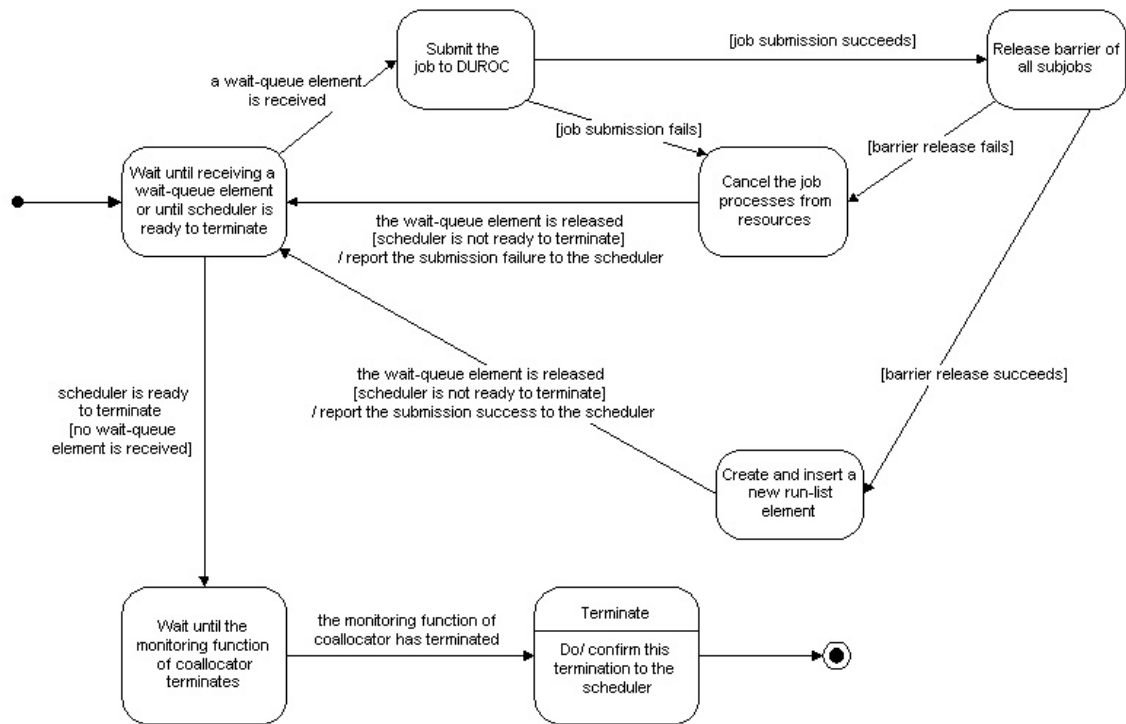


Figure 4.4: The state diagram of the Co-allocator for job submission

Since the Co-allocator must submit another job to DUROC while it also has to monitor the progress of running jobs, the whole activity of the Co-allocator for job monitoring must be implemented as another single thread. The monitoring activity can be summarized in the state diagram in Figure 4.5.

4.4 Communication and Synchronization between Threads

From Section 4.3, we know that there are at least four single threads involved in this co-allocation service. The Resource Monitor provides one thread to check out periodically whether there is a change in resource availability since a (re)scheduling failure occurs (i.e., a job cannot fit the requested resources). The Scheduler provides another thread to carry out its activities. The Co-allocator provides two separate threads, for job submission and for job monitoring. Including the thread of the main program which starts all of the four threads, called the *main* thread, we can describe the communication and synchronization between them in an adapted event-trace diagram in Figure 4.6.

In the diagram, there are three pairs of threads which implement producer-consumer problem [15]: {main, scheduler}, {co-allocator(submission), co-allocator(monitored)}, and {co-allocator(monitored), scheduler}. For all of the pairs, the left side plays a producer, and the right side plays a consumer. Other pairs, {scheduler, resource monitor} and {scheduler, co-allocator(submission)}, implement common synchronization scheme.

In the {main, scheduler} communication, the scheduler thread will be disabled until the main thread puts an element to the wait-queue, which is their shared resource. Vice versa, while the scheduler thread has access to the wait-queue during a (re)scheduling activity, the main thread is not allowed to access the wait-queue. When the scheduler finishes its (re)scheduling activity and no longer needs the wait-queue for a while, it must release its access before continuing to subsequent activities, in order to enable the main thread again. Therefore, while the scheduler deals with the co-allocator(submission) thread, several elements can be appended into the wait-

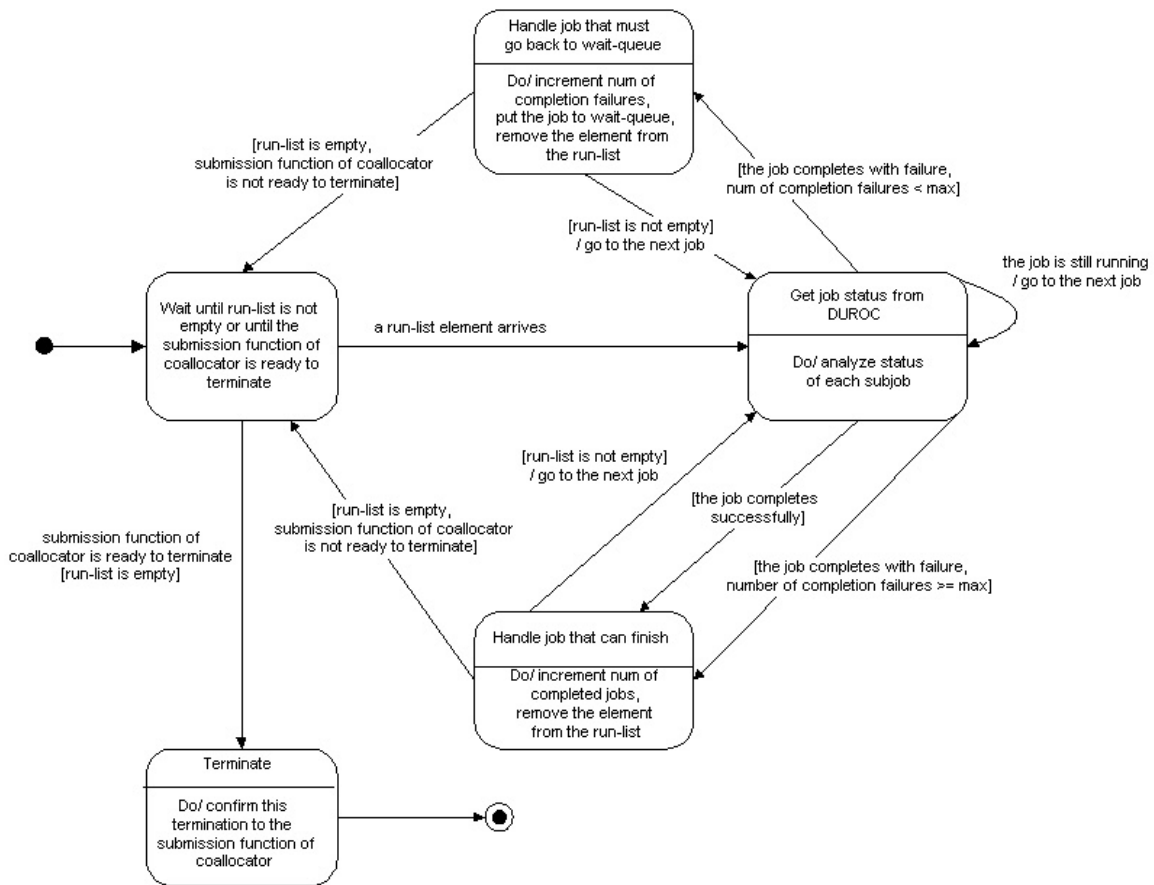


Figure 4.5: The state diagram of the Co-allocator for job monitoring

queue by the main thread.

When the scheduler thread sends a scheduled job to the co-allocator(submission) thread, the scheduler is disabled, waiting for the co-allocator(submission) thread to finish the submission. Suppose the submission is successful, then the coallocator(submission) puts a new element to the run-list, which is the shared resource for {co-allocator(submission), co-allocator(monitng)}. Co-allocator(monitng) will only starts its activities when the co-allocator(submission) releases its access to the run-list. When the co-allocator(submission) thread sends the signal that it has finished, the scheduler can continue its activity to get another element from the wait-queue and do a (re)scheduling again. We can see that while the scheduler gets back to work, the co-allocator(monitng) can also work at the same time. The same scenario will apply in case a (re)scheduling failure occurs, in that the scheduler must wait until the resource monitor sends a release signal.

Sometimes, when there is a job completed, the co-allocator(monitng) sends the current number of completed jobs to the scheduler. We can see in the diagram, that the sending event takes place when the co-allocator(monitng) is keeping the access to the run-list. It means, the scheduler cannot send any job to the coallocator(submission) at the same time when it is receiving the number of completed jobs.

It may also happen that a job completes with failure, and the co-allocator(monitng) puts the job back to the wait-queue. Similar to the event when the main thread puts an element to the wait-queue, during this event, the scheduler cannot access the wait-queue.

When a thread finishes all of its activities, it cannot terminate immediately, but first it must send a signal indicating that it is ready to terminate to another thread. Later on, after the other thread confirms its termination, the former thread may terminate as well. In this co-allocation

service, which thread waits for which one to terminate is described clearly in Figure 4.6. This order also conforms to the state diagrams depicted in Figure 4.3, Figure 4.4, and Figure 4.5.

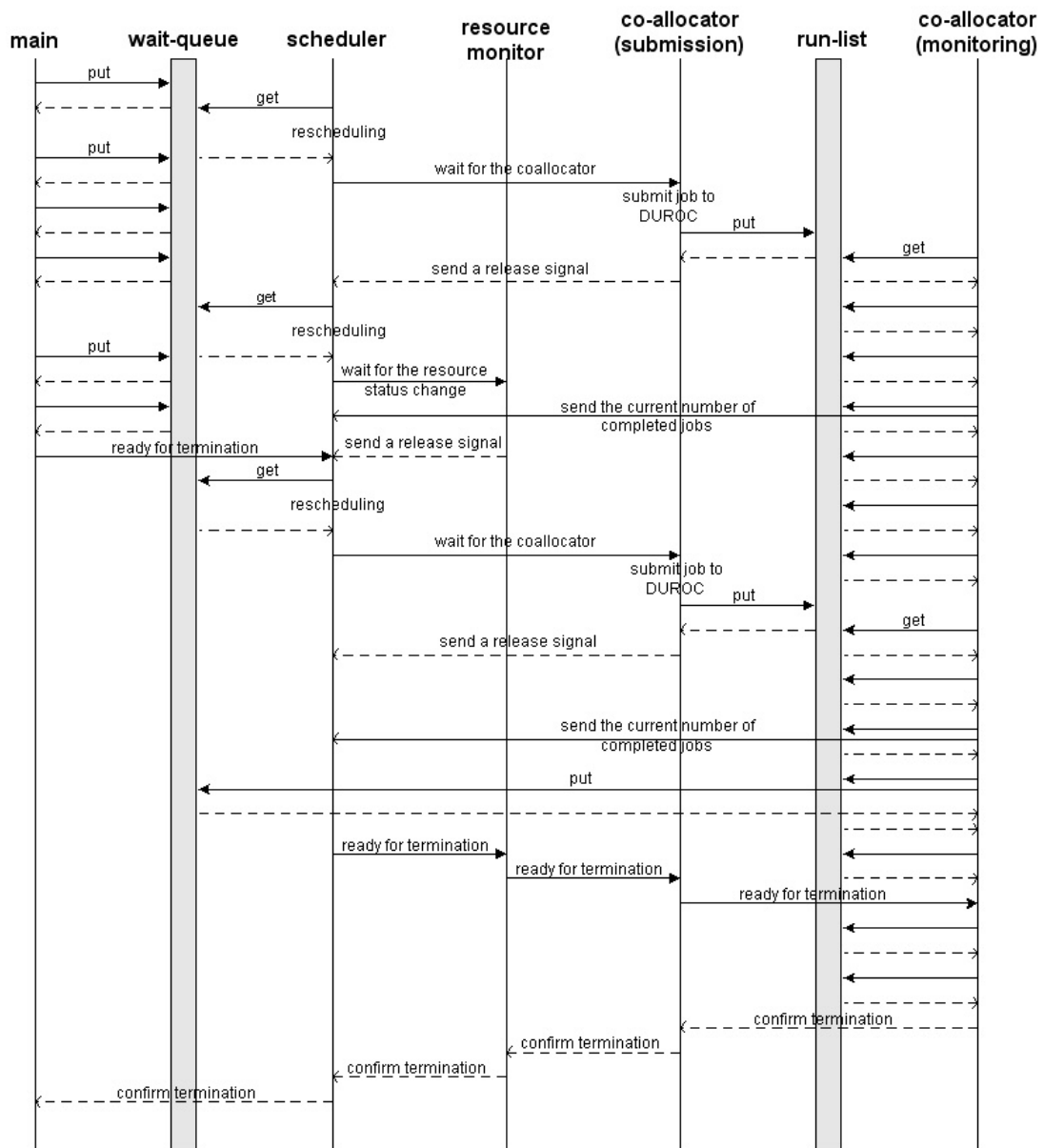


Figure 4.6: The adapted event-trace diagram of communication and synchronization between threads in DCS

Chapter 5

Implementation of the Co-allocation Service

In this chapter, we will focus on the implementation issues regarding the DCS, whose design was already discussed in Chapter 4. The service is implemented as a multithreaded program written in C language. First, we will discuss the structure of modules that constructs the DCS, including data structures used in each module. We also discuss how some supportive utilities such as GASS are implemented.

5.1 Modules and Data Structures

The DCS comprises several modules which can be classified as:

1. Globus modules: *globus_common*, *globus_rsl*, *globus_duroc_control*, *globus_gass_server_ez*, and *globus_gram_client*.
2. Support modules: *io_mgmt*, *thr_mgmt*, and *list_mgmt*.
3. Main modules: *resmonitor*, *broker*, *scheduler*, *coalloc*, and *dcs*.

The relations between the modules are depicted in Figure 5.1 as arrows. Every relation refers to a file inclusion, the module at the arrow butt including the module at the arrow head through the `#include` directive.

Each of these modules can be represented either by a `.c` (source) file or `.h` (header) file, except the *dcs* module, which has the source file. The header files are needed to publicize external data structures and functions that will be required by other modules. The module from which a dashed line comes out must be represented as a source file, while the module from which a solid line comes out must be a header file. However, the module at the arrow head (i.e., the included module) must always be a header file. Every source file, except *dcs.c*, will always include its own header file.

As an example, we can conclude from Figure 5.1 that *coalloc.c* includes *coalloc.h* which is its own header file, *coalloc.h* includes *scheduler.h*, and *scheduler.h* includes *list_mgmt.h*. Therefore, *coalloc.c* basically includes *list_mgmt.h*. However, since *scheduler.c*, instead of *scheduler.h*, is the module that includes *broker.h*, *coalloc.c* does not include *broker.h*.

5.1.1 Globus Modules

Every Globus API function called in this application is provided by a Globus module. We will not give details about every Globus module here for we can get them from the Globus website. Suffice it to say a few relevant things about some Globus modules utilized in the DCS.

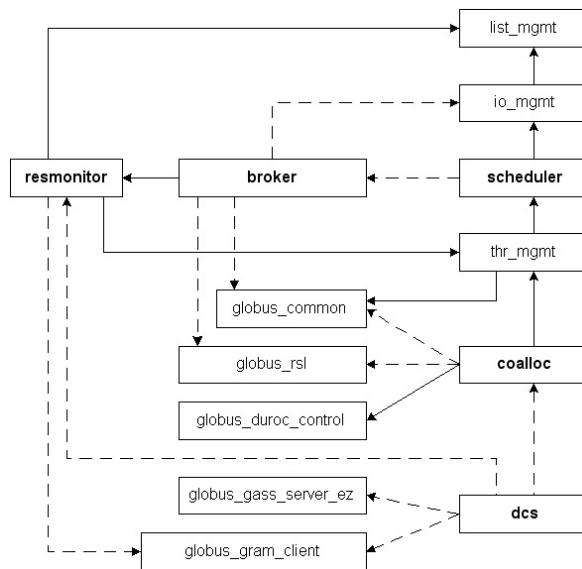


Figure 5.1: The structure of modules in the DCS

Most of the Globus modules utilized to support this application must be activated in the main function before all other activities are started and must be deactivated after all other activities have been completed. Only *globus_rsl* does not have to be activated since its activation has been included in the *globus_common* module.

The *globus_common* module must be included by every module that requires thread utilities, since this application uses thread model provided by the Globus Toolkit. In this case, all of the main modules will include *globus_common*.

The *globus_rsl* module is required by main modules that need to parse an RSL string or process an RSL tree. However, it must be accompanied by *globus_common*. In fact, its inclusion must be preceded by the *globus_common* module's inclusion. In this co-allocation service, the *globus_rsl* module is included by *broker* and *coalloc* modules.

The *globus_duroc_control* module is required especially by *coalloc* module which submits jobs to Globus DUROC and monitors their progress. However, since *dcs* module must activate and deactivate the *globus_duroc_control* module, it also requires the respective module.

The *globus_gass_server_ez* module is included only by *dcs* module because the activation and deactivation of GASS server is done there.

The *globus_gram_client* module is included by the *resmonitor* module because it provides a Globus function which can detect clusters that are currently disconnected or disabled before building the idle-list, so such clusters will not participate in the resource brokering. The *dcs* module also include this Globus module because it is responsible for activating and deactivating the Globus module.

5.1.2 Support Modules

The support modules are necessary to provide basic facilities such as input/output management, thread management, and list management. These facilities will be shared by other modules, especially the main modules.

As described in its name, the capabilities provided by the *io_mgmt* module encompasses reading a text file and storing all its content into a single string of characters, writing the content of a string of characters into a single text file, and writing to a text file whatever written to the standard output without just redirecting the standard output. The file reading capability is then utilized to read a batch file containing a list of job request file names that will be submitted to

the *dcs*. It is also utilized to read a single job request file (i.e., ordered and unordered type) to produce an RSL string. An RSL string is the form of a job request when it is submitted to the Globus DUROC. It is passed as a parameter in the *globus_duroc_control_job_request()* function.

Among the main modules, *scheduler*, *coalloc*, and *dcs* modules are the ones which require the *io_mgmt* module. The *scheduler* module obviously needs it when reading and writing job request files. The *coalloc* module uses it for the purpose of writing a log file when monitoring the progress of running jobs. The *dcs* module requires it especially to read a batch file mentioned previously, extract the list of file names, and give them to the Scheduler to be processed.

The *thr_mgmt* module is meant to define and initialize global variables which are shared by different modules in DCS. Some of them are necessary in the communication and synchronization of threads, but others will just be shared by different modules which are in the same thread of control. Since the threads themselves are created in the *dcs* module, *thr_mgmt* is like a header file for *dcs* module. With these global variables, we no longer need to define thread-specific data which is more complicated. The *thr_mgmt* module also provides functions to simplify the termination of threads in this co-allocation service.

The global variables are enclosed in a structure called *thread_data_t* to enable them passed through various threaded functions. They are categorized into four parts:

- Common variables shared by several modules or required in the communication and synchronization between threads. They are listed in Table 5.1.
- Boolean variables indicating whether to wait for a thread (TRUE) or not (FALSE). This is important in the termination of threads. Such variables are *waitSchedRun*, *waitResSpy*, *waitCoallocSubmit*, *waitCoallocMonitor*, *waitSurveillance*.
- Mutex and conditional variables needed to manage mutual exclusions and synchronization between threads. Such variables are *submitMutex*, *monitorMutex*, *controlMutex*, *schedRunMutex*, *resSpyMutex*, *surveillanceMutex*, *submitAllowed*, *monitorAllowed*, *schedRunAllowed*, *successAllowed*, *resSpyAllowed*, *surveillanceAllowed*.
- The identities of threads given for their creation and required in their termination as well. Such variables are *thrCoallocSubmit*, *thrCoallocMonitor*, *thrSchedRun*, *thrResSpy*, *thrSurveillance*.

The *list_mgmt* module defines the basic structure for every linked list running in the DCS. The wait-queue managed by the scheduler is also based on this structure. It also provides some primitives to manage the list, e.g., inserting a new element to a list in a certain order, deleting an existing element in any position from a list, etc.

Every list element contains *datum* data field that can be assigned with any pointer to any data structure, as described below. This is why the element structure can be implemented for any linked list.

```
typedef struct list_tag {
    void* datum;
    struct list_tag* next;
} list_t;
```

The *list_mgmt* module is required by most of the main modules, i.e., by *coalloc* to create and maintain the run-list, *scheduler* to create and maintain the wait-queue, and *broker* to manage the request-list, and *resmonitor* to manage the idle-list.

5.1.3 Main Modules

Every main module, except *dcs*, correlates directly with a main component described in Section 4.3. Their names clearly describe which main components they implement. The *dcs* module

Table 5.1: Common variables shared by several modules

Var Name	Type	Description
submitData	submit_data.t	The pointer to an element of wait-queue currently needed by the Co-allocator.
fileNames	list_t*	A list of file names, useful as a buffer for all user job requests before they are inserted to the wait-queue.
idleHead	list_t*	The pointer to the head element of idle-list.
queueHead	list_t*	The pointer to the head element of wait-queue.
jobStartSuccess	int	A boolean indicating whether this job is submitted successfully (TRUE) or not (FALSE).
endCount	int	The number of completed jobs up to this time.
removedCount	int	The number of removed jobs due to failures.
gassURLString	char*	The URL address of a GASS server installed currently.
options	unsigned long	The menu option to run DCS (e.g., activating worst-fit or bestfit, FCFS or FPFs).
runResSpy	int	A boolean indicating whether to run the resSpy thread (TRUE) or not (FALSE).
runResMonitor	int	A boolean indicating whether to invoke the Resource Monitor (TRUE) or not (FALSE).

itself is made to contain the main function. Next, we will discuss each of these modules to know some specific things concerning the implementation of the main components of DCS.

The *resmonitor* Module

This module implements the functionalities of the Resource Monitor. It defines a structure for each element of an idle-list, the buffer to store the numbers of idle processors of all clusters. It is called *idle_elt.t* which is described in Table 5.2.

Table 5.2: The structure of an idle-list element

Var Name	Type	Description
clusterID	int	The identity of this cluster (e.g., 0 for fs0, 1 for fs1).
nodeNum	int	The amount of processors belonging to this cluster.
checked	int	A boolean indicating whether this cluster has been used by a subjob (TRUE) or not (FALSE) in an attempt to fit the corresponding job.

Idle-list may have more than one instance. What has been discussed in Section 4.3.3 about a data structure used by the Resource Broker to fit a job request, is an idle-list instance as a global variable. In the implementation, it is called *idleHead* and it is shared by this module and the *broker* module (also listed in Table 5.1).

The function to build an idle-list is called *build_idlelist()*. It invokes a Perl script through a pipe mechanism to build the idle-list instance. Perl is well known for its capability of regular expression, so the Perl script can help manipulating symbols or characters resulted from retrieving resource status data from PBS. The *build_idlelist()* function is not a thread and will be called by a function in the *broker* module to activate the Resource Monitor.

This module also provides a functionality to handle a (re)scheduling failure (i.e. a job cannot fit its requested resources) mentioned in Section 4.3.4. It is implemented as a threaded function called *res_spy()*, which queries the numbers of idle processors in every cluster periodically from PBS and signal a function in the *scheduler* module when there is a change in resource status to

activate (re)scheduling again. The *res_spy()* function builds its own idle-list. This is an example of building an idle-list instance as a local variable.

The *resmonitor* module also provides other functions to manipulate an idle-list, such as copying, removing, and displaying an idle-list instance.

The *broker* Module

The *broker* module implements the functionalities of the Resource Broker. An important data structure related to Resource Broker is the request-list. Each of its elements is implemented as a structure called *req_elmt_t*, each of which contains data-fields listed in Table 5.3.

Table 5.3: The structure of an request-list element

Var Name	Type	Description
nodeNum	int	The number of processors belonging to this cluster.
cluster	idle_elmt_t*	The pointer to an element of idle-list.
subjob	globus_rsl_t*	The pointer to an element of RSL tree.

It also defines an external function called *run_broker()* to activate the Resource Broker, starting from building the request-list through matching its elements to the elements of idle-list. The request-list itself is declared as a local variable in this function. This function is not a thread and will be called by a function in the *scheduler* module when the (re)scheduling activity takes place. When this function needs the Resource Monitor, it actually calls *build_idlelist()* which is defined in the *resmonitor* module.

In case FPFS is employed, *run_broker()* can be invoked multiple times in a (re)scheduling activity. However, since the Resource Monitor can only be invoked once during a (re)scheduling activity, *run_broker()* needs to prohibit the calling of *build_idlelist()*. This can be done through a flag called *runResMonitor* which is listed in Table 5.1. This variable is shared between *broker* and *scheduler* modules.

The *scheduler* Module

The *scheduler* module provides the functionalities of the Scheduler component. It defines a structure called *submit_data_t* to implement the element of the wait-queue. Table 5.4 lists all the fields of the structure.

The wait-queue itself is implemented as *queueHead* which is also a global variable, listed in Table 5.1. The function to activate the Scheduler is called *sched_run()*, where the (re)scheduling and element processing activities take place. It is a thread, and it needs the global variables declared in the *thr_mgmt* module. This thread is the one which communicates most often with other threads.

When a new element is inserted to the wait-queue, the *startTime* field of the element is set up with the current value of time. The field value can be modified when the element is inserted again to the wait-queue due to a submission or completion failure.

The *coalloc* Module

This module implements the functionalities of the Co-allocator, both job submission and job monitoring. It also defines a data structure which is called *monitor_data_t* that implements the element of the run-list, as described in Table 5.5.

The run-list itself is implemented as a local and static variable called *listHead*. The job submission activity is implemented in the *coalloc_submit()* function, while the job monitoring

Table 5.4: The structure of a wait-queue element

Var Name	Type	Description
jobID	int	The identity of this job during its life in the DCS.
fileName	char*	The file name of this job request.
jobType	enum jobTypeDef	An enumerated integer to indicate a job type (ordered or un-ordered).
originStr	char*	The text of original job request (probably still in the incomplete RSL string).
jobReqStr	char*	The complete RSL string for the job request.
startFailNum	int	The number of submission failures suffered by this job up to this time.
endFailNum	int	The number of completion failures suffered by this job up to this time.
subjobCount	int	The number of components (subjobs) in this job.
subjobLabels	char**	An array of <i>label</i> parameters of all subjobs in this job.
subjobReqNums	int*	An array of requested number of processors of all subjobs in this job.
subjobClustIDs	int*	An array of identities of clusters allocated for all subjobs in this job.
startTime	struct timeval	A point of time when this job is inserted to the wait-queue.
serviceTime	struct timeval	A point of time when this job is submitted to DUROC.

Table 5.5: The structure of a run-list element

Var Name	Type	Description
submitData	submit_data.t*	The pointer to an element of wait-queue associated to this job.
jobContact	char*	An identity given by the Globus DUROC for this job during its execution.
subjobStates	int*	An array of current status of all subjobs in this job.
preSubjobStates	int*	An array of previous status of all subjobs in this job.
completed	int	A boolean indicating whether this job has completed (TRUE) or not (FALSE).
pending	int	A boolean indicating whether this job has been pending (TRUE) or not (FALSE).
latestChangeTime	struct timeval	The latest point of time when a change occurs in this job status.

activity is implemented in the *coalloc_monitor()* function. Both functions are executed in separate threads.

The time when the *coalloc_submit()* function submits a job request to DUROC is called service time, and it is recorded in the *serviceTime* field of the corresponding wait-queue element.

The *coalloc_monitor()* function implements the status polling for every job in the run-list which is already mentioned in Section 4.3.5. The function always records the time at the moment before it polls the first job in the run-list.

In every polling, the current status of all subjobs is stored in the *subjobStates* field, and the *coalloc_monitor()* function will compare it to the *preSubjobStates* field which stores the status of the same subjobs in the previous polling of the same job. If the current status is exactly the same as the previous one, the current status will not be recorded. The current status will be recorded if there is a difference from the previous one.

If the previous and current status of a job is the same and the status is *pending*, the *coalloc_monitor()* will check whether the job is already in pending status for a period that exceeds a configurable maximum length of time. If so, the *coalloc_monitor()* will count the case as a

completion failure, and do the same procedure to the job as to those of the same failure.

The *globus_duroc_control_job_cancel()* function that is called when there is a submission or completion failure sometimes causes the callback message about the corresponding job's status incorrect. In that case, the whole DCS system will get an abort signal which ultimately bring the whole DCS terminated abnormally. Unfortunately, in this current implementation, there is no error handling against this dissappointing occurrence. However, there is possibility to redirect the effect of the abort signal in various tricky ways.

The *dcs* Module

This module contains the *main()* function, so it is basically the first module to start when the DCS is run. It initializes a global data structure, activates and deactivates some Globus modules as well as the GASS server, manages switches/options for the command line as its user interface, and creates all of the threads used in this co-allocation service.

It also puts all the user job requests into the wait-queue. A user can input the job requests at once in a batch file or in separate job request files. The types of the job requests will be known simply by their extensions. The *.rsl* extension is for ordered jobs, while *.uno* is for unordered jobs. At the moment, there is no further validation effort towards the types of job requests.

All the user job requests will be stored first in a buffer called *fileNames* which is listed in Table 5.1. One by one, each job request will be inserted to the wait-queue.

5.2 Additional Supportive Utilities

Some utilities added to this co-allocation service are the GASS server and *surveillance* thread. The GASS server is implemented to enable users to see the results (output files, error files) from their job execution, such as the time measurement, etc. In DCS, user-executable hosts and submission hosts are always the same machine, a specific case already mentioned in Section 2.2.2.

The implementation of GASS server will be carried out by DCS so that users do not have to rewrite RSL files or some programming every time they need a GASS service. Users can specify their job requests in *.rsl* or *.uno* files without having to copy the executables to every destination cluster. However, for executables that require input files, the users must copy the input files to every destination cluster, and record their locations in the RSL *directory* parameters in the job requests. The reason for this is the facility to stage input files has not been implemented in DCS for now. For executables that do not need any input file, the users can omit the *directory* parameters altogether from the job requests.

All of the output files and error files will be staged back to the submission hosts, in a sub-directory called *outputs*, which is located in the same directory where DCS resides.

The implementation of GASS server in DCS is quite simple [9]. Before all threads of DCS are started, the DCS calls *globus_gass_server_ez_init()*, a function provided by Globus to start a local GASS server. After the Co-allocator receives a job request from the Scheduler, the Co-allocator adds the value of *gassURLstring*, a global variable listed in Table 5.1, to the job request. The Co-allocator does it for all job requests it receives before it submits them to DUROC.

The surveillance thread is implemented to detect periodically the system load of every DAS cluster. Its function is added in the *resmonitor* module. Like *build_idlelist()*, it invokes a Perl script to retrieve the number of busy processors status from PBS in each cluster. In this case, however, the surveillance thread calculates the system load not only due to all users' jobs but also due to DCS jobs. The surveillance thread is also started in *dcs* module, and it is terminated when all other threads, except the main thread, have been terminated. Afterwards, the main thread itself will terminate.

Chapter 6

Experiments with the Co-allocation Service

This chapter talks about some experiments that have been done with the DCS. Its purpose is to show that the DCS can work correctly as designed, so the experiments are not intended for any complex performance analysis. However, with those experiments, we can also observe the influence of decisions made to job types, job sizes, and scheduling policies, to the general performance of the whole system as already studied extensively in previous works [2, 4]. Therefore, we still need to measure some parameters, which are the job response time (the total time since the job is submitted to DCS until the job is completed), job run time (the time spent by the actual processes to run in their allocated processors), and the time due to DUROC's overhead in submission and completion of the job.

For each experiment, the system load (utilization) due to our own jobs and due to other users' jobs in the involved clusters are shown in a diagram based on what has been detected by the surveillance thread mentioned in Section 5.2. This information describes the background load during the time of an experiment.

The selected test cases encompass differences in usage of job sizes, job types, and scheduling policies, and also the number of clusters (hence, number of processors) that are actually used in a single experiment. We choose the Poisson application [1] as the application program (executable) for all the experiments. It is eligible to implement co-allocation because it contains significant parts of communication between processors. Next, we will give a brief explanation about the Poisson application, and afterwards discuss all the experimental results.

6.1 The Poisson Application

The application used for the experiments implements a parallel iterative algorithm to solve the Poisson equation with a red-black Gauss-Seidel scheme. It searches for a discrete approximation to the solution of the two-dimensional Poisson equation, that is using a second-order differential equation governing steady-state head flow in a two-dimensional computational domain. The domain is a unit square or grid consisting of black and red points, with every red point is having only black neighbours and vice versa.

When the application runs, the whole unit square is split up into uniform rectangles, each of which is for a single participating process. Each process runs in parallel with others doing a number of iterations until a global stopping criterion is fulfilled. In every iteration, the value of each grid point is updated as a function of its previous value and the values of its four neighbours, and all points of one colour must be visited prior to the ones of the other colour.

Every process communicates with its neighbours to exchange the values of grid points on the borders of their rectangles and to compute a global error which is useful for the stopping criterion.

Exchanging values in the borders is done in the following order: first in the top borders, then bottom, left, and right.

When the Poisson application is executed in multiple clusters, the process grid is divided into adjacent vertical strips. Figure 6.1 shows an example of the even division of processes into two clusters.

3	7	11	15	19	23	27	31
2	6	10	14	18	22	26	30
1	5	9	13	17	21	25	29
0	4	8	12	16	20	24	28

Figure 6.1: The process grid for 8x4 configuration evenly divided over 2 clusters

The process grid configuration is defined as the number of processes in horizontal direction times the number of processes in vertical direction. Each process maintains a rectangle consisting of a set of grid points. In Figure 6.1, since the process grid is evenly divided over 2 clusters, each cluster will have 16 processes in it. Such a distribution of processes can also be declared as 2x16 cluster configuration. However, the cluster configuration is associated to the job size and therefore it should not be confused with the process grid configuration.

6.2 General Conditions for the Experiments

There are five experiments that will be discussed here. In all of the experiments, we submit a batch of 40 jobs of the Poisson application to DCS. All of the jobs arrive in the Scheduler at nearly the same time, so there is no inter-arrival time among them. In all the experiments but one, we submit either ordered or unordered jobs. In the remaining one, we submit an even mix between both of them. Every job has 4 components (subjobs) of equal size which is either of 4x8 cluster configuration or 4x4 cluster configuration. In these experiments, there is neither submission failure nor completion failure, so there is no job removed due to those failures, but rescheduling failures might be indicated in the diagrams.

Only one of the experiments uses 4 clusters of the DAS, namely fs0 which has 144 processors, and three other clusters (fs1, fs2, and fs3), each of which has 64 processors. In all the other experiments, only two clusters, fs0 and fs3, could be used because of some instability exhibited by the other clusters at that time.

For each experiment, a diagram is provided to describe the system load caused by DCS jobs and other users' jobs during the time of the experiment. The system loads are normalized with regard to the clusters that are actually used.

6.3 Experimental Results

In the first experiment, 4 clusters are employed, all jobs are unordered, each of which has 4x8 cluster configuration, and the scheduling policy is FCFS. The result is shown in Figure 6.2.

We can see immediately that the DCS jobs considerably increase the total system load. The sudden drops in the system load due to DCS jobs indicate that some jobs depart while new jobs have not been submitted to their allocated resources. The delay of job submission is caused by resource insufficiency detected by the Resource Monitor at that moment, so that the Scheduler is disabled, waiting for the Resource Monitor to notify it when there is a change in resource

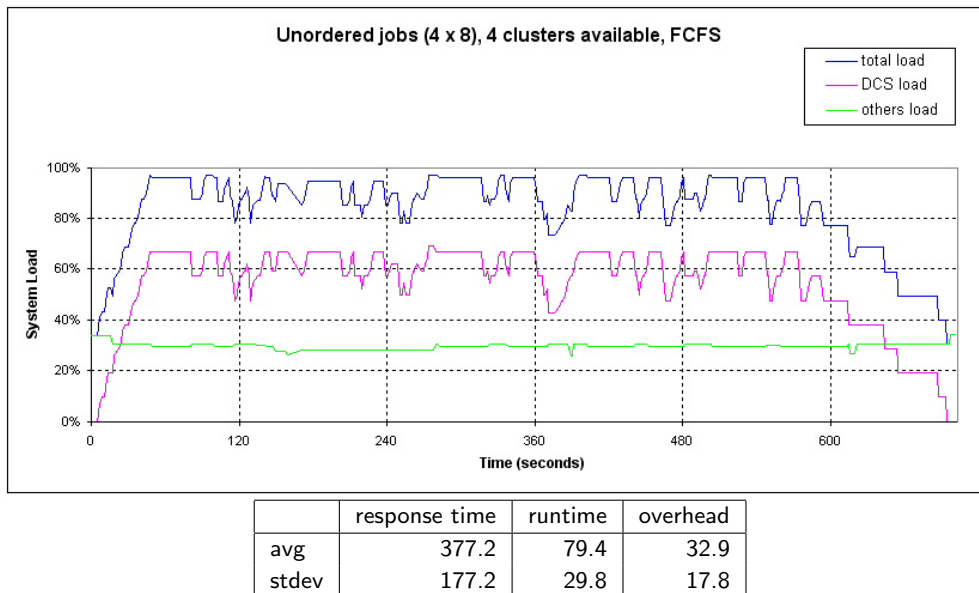


Figure 6.2: The first experiment with 4 clusters (144 + 3x64), unordered jobs of size 4x8, and FCFS.

availability. However, it takes a few seconds before the Resource Monitor notices the change. This situation leads to the delay of the submission of new jobs.

The flat segment in the system load curve due to DCS jobs indicate there are no sufficient resources during that period of time, but in this case, no currently running jobs complete and depart from the system. Therefore, both declining and flat segment in the system load curve imply that subsequent jobs are suspended in the wait-queue until there are sufficient resources for them. Based on the flat segments in the curve, we can calculate the maximum number of DCS jobs that can be executed at the same time. In the first experiment, the system load lies at 66.67% level for the flat segments, meaning that only 224 processors at most can be allocated at the same time for DCS jobs. Since all the DCS jobs have equal size, at most 7 jobs submitted by DCS can be executed in those four clusters at the same time.

The second experiment (Figure 6.3) employs the same conditions as those of the first experiment (Figure 6.2), except the number of clusters that are actually used. In this experiment, only 2 clusters are employed: the one with 144 processors, and the other one with 64 processors. The condition of using fewer clusters (in fact, fewer processors) increase the duration of the experiment and also the average response time, compared with the same parameters of the first experiment. This situation is also supported by the maximum number of DCS jobs that can be executed simultaneously. With the same way of calculation, we get the result that at most 5 jobs submitted by DCS can run in their allocated clusters at the same time. Consequently, it takes more time to complete all of the 40 DCS jobs.

The third experiment shown in Figure 6.4 has the same conditions as those of the second experiment (Figure 6.3) but one difference, which is the job size. In this experiment, every job has 4x4 cluster configuration which is smaller than 4x8. The total duration of the second experiment is longer than that of the third experiment due to the larger job size but also to the longer average runtime. The difference of total duration can also be explained through the flat segments that are much more in the second experiment than those in the third experiment. It implies that jobs in the second experiment are more often to wait for resource availability than those in the third experiment.

The fourth experiment (Figure 6.5) still uses the same job size and scheduling policy as those of the second experiment (Figure 6.3). However, it uses ordered jobs rather than unordered

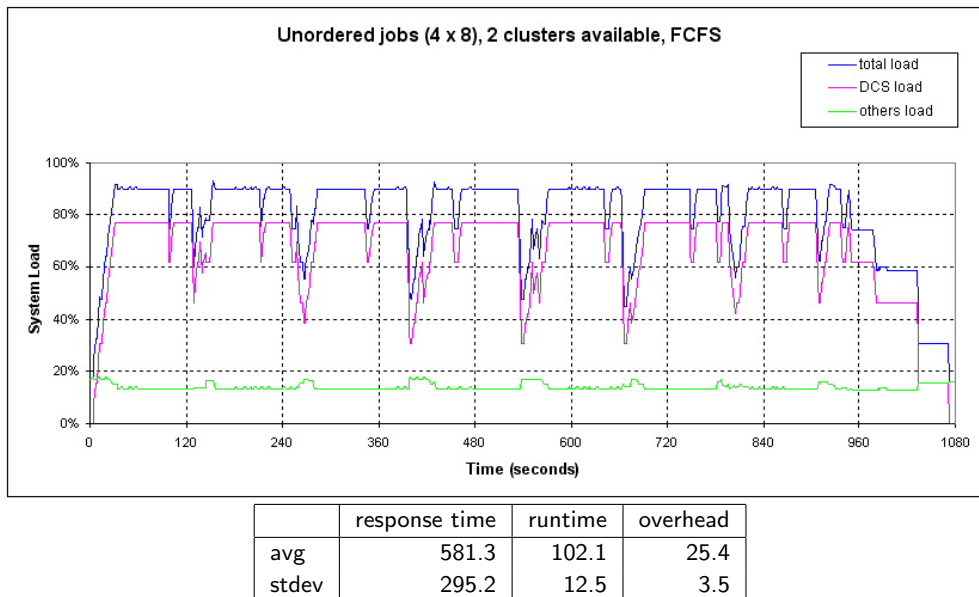


Figure 6.3: The second experiment with 2 clusters (144+64), unordered jobs of size 4x8, and FCFS.

jobs. It is important to note that in this experiment, two components go to the cluster with 144 processors and the other two go to the cluster with 64 processors. It would have the same situation as that of 2x8 cluster configuration. The even and fixed distribution of components to their allocated clusters while the clusters have different number of processors (i.e., 144 and 64) makes the cluster with larger number of processors cannot be utilized more although there is still space for that. As the result, it is quite often to occur that jobs are suspended in the wait-queue until currently running jobs depart from the system after their completion. It is indicated by too many flat segments in the DCS system load curve. This situation makes total duration and average response time in the fourth experiment become significantly longer than those in the second experiment which employs unordered jobs.

The fifth experiment (Figure 6.6) employs FPFS as the scheduling policy. In addition, a half of the jobs are ordered and the rest are unordered. All other conditions are the same as those of the fourth experiment. What makes interesting here is the total duration and average response time are improved as compared to the fourth experiment. It is caused by the presence of unordered jobs and FPFS so that when an ordered job is suspended due to disapproval from the Resource Broker, the unordered job can get priority and submitted to its destination clusters. In case FCFS is employed in the fifth experiment, the result would not be much different than that of the fourth experiment.

All the experiments show that the standard deviation of average response times is quite large. This is due to the arrival of all jobs (in the Scheduler) in a single experiment is almost at the same time. Also, we can see that the overhead time of DUROC is quite large. It is caused by the initialization that DUROC does when it receives a job request, such as decomposing into multiple subjobs, sending each subjob to GRAMs, and wait the callback message from GRAMs because the submission function is blocking (synchronous). The overhead is also caused when a job completes, because it takes some time before DUROC gets the most recently status of each subjob.

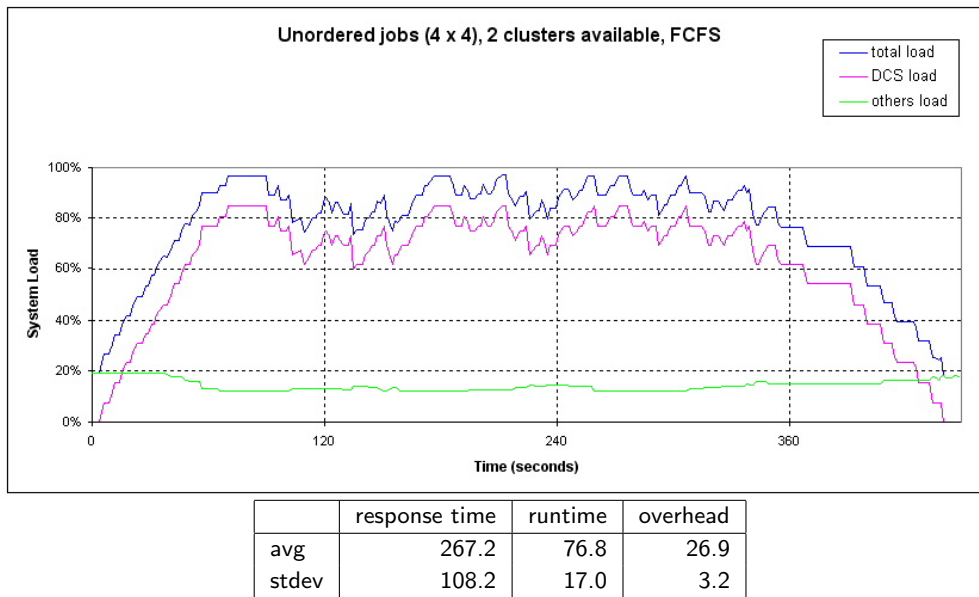


Figure 6.4: The third experiment with 2 clusters (144+64), unordered jobs of size 4x4, and FCFS.

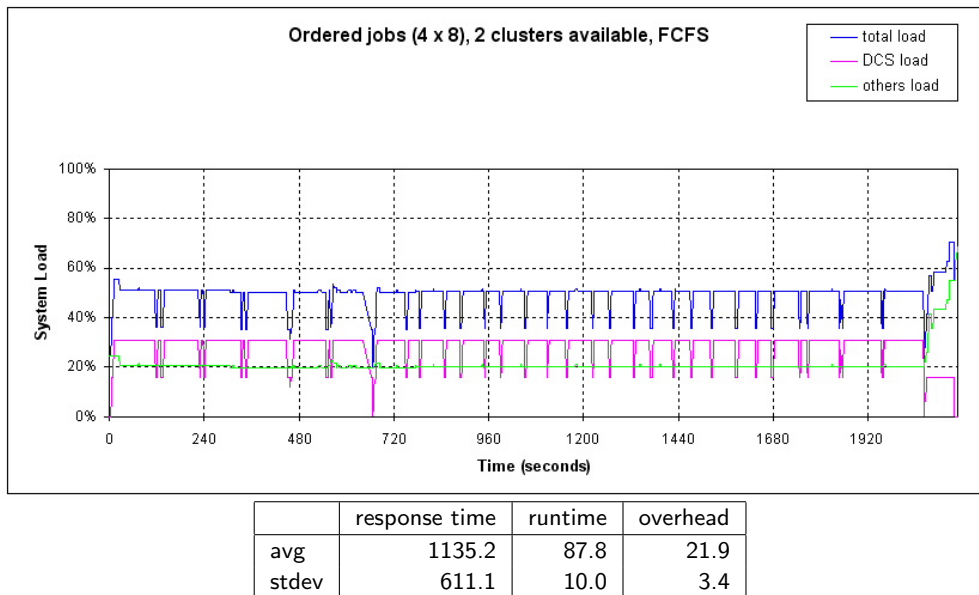
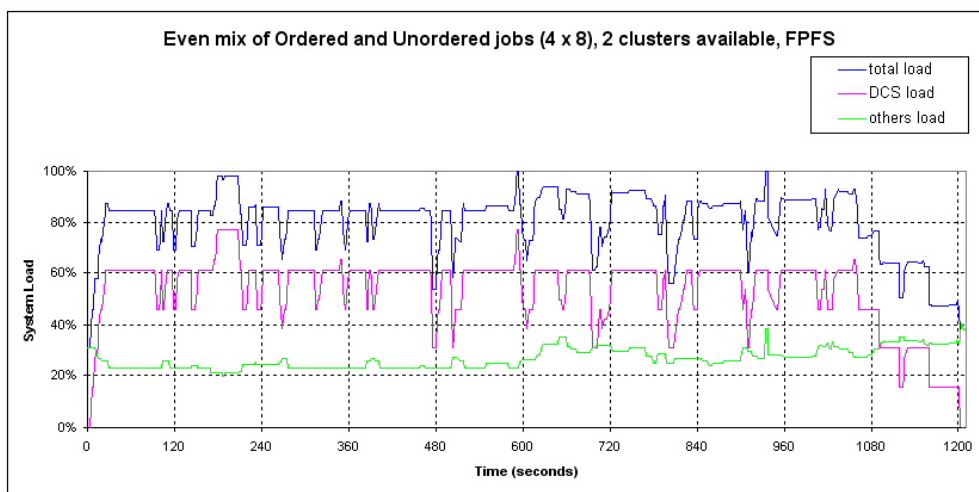


Figure 6.5: The fourth experiment with 2 clusters (144+64), ordered jobs of size 4x8, and FCFS.



	response time	runtime	overhead
avg	634.7	92.4	25.3
stdev	335.3	15.6	6.5

Figure 6.6: The fifth experiment with 2 clusters (144+64), an even mix of ordered and unordered jobs of size 4x8, and FPFS.

Chapter 7

Conclusions and Future Work

In this report, we have presented the design and implementation of the Dynamic Co-allocation Service (DCS) for processor allocation in multicluster systems, using DAS as its testbed. We have also shown the results of experiments that shows DCS working correctly as designed, and that it is able to achieve a quite high total system load, although the jobs submitted in our experiments were not large (max. 32 processors). The results of the experiments also shows that indeed developing such a dynamic mechanism for co-allocation, along with appropriate job-fitting and scheduling strategies, can improve the performance of executing user jobs in multicluster systems.

Definitely there are still many things to do that can improve this co-allocation service. One of them is adding a capability to handle abort signals that possibly come when there are callback message errors, caused by instability of the involved clusters, or by inappropriate termination of job processes while they have not completed. Such a handle is expected to lead the DCS to cancel the submitted jobs safely although the whole DCS might be imposed to stop. In other words, the DCS would not be just aborted dissapointingly when such attacks come.

Another thing that is useful is designing and implementing the format of job requests for all types (not only for ordered and unordered jobs). Flexible jobs should be considered to be employed, but the format for that type of job cannot be just written in RSL and lack some of its parameters. There should be a format that can cover all types of job requests, and XML is one of the best options, because XML is more flexible and portable than RSL.

This co-allocation service is only at the beginning of our design and implementation effort of co-allocation in grids. In particular, there are plans to extend the current design of the DCS to more types of resources, to more heterogeneous systems both with respect to the hardware and the local resource managers, and to more complicated job types (e.g., work flows). Therefore, the use of Globus Toolkit 3 (GT3) as the middleware that supports this co-allocation service should be considered. Using GT3, this co-allocation service could become one of the fundamental grid services, which is very useful in grids, not only in multicluster systems.

We note some efforts to design mechanisms for the co-allocation of both processors and information resources in which DUROC will be replaced altogether and build co-allocation mechanisms on top of separate Globus components. Nevertheless, we also note that DUROC has been doing its job very well, especially in coordinating the communication between multiple actual processes of the same job in different clusters, despite its lacking of other things like resource brokering and complete fault tolerance.

Finally, it is always worth to do a better performance analysis. One of the complicating factors here is the lack of reproducibility of experiments in systems that have a background load submitted by other users that we cannot control.

Appendix A

User's Manual for DCS

This document is a guide to install, compile, and use the Dynamic Co-allocation Service (DCS). In order to utilize DCS, a user must have an account on the DAS and a Globus user certificate. Throughout this document, the user prompt will be displayed as '\$'.

A.1 Installing and compiling the DCS package

The DCS package is stored in the file *dcx-package.tar.gz*, and it can be installed in any DAS cluster and in any directory where the user has access, through this command:

```
$ tar -xvf dcs-package.tar.gz
```

After unpacking the package, the following directories will be created:

- *dcx-source*, the collection of source files, header files, and other supporting files of the DCS.
- *poisson-case*, the collection of job requests for the Poisson application along with its source codes and executables.
- *cpi-globus-case*, the collection of job requests for the application of calculating pi number along with its source codes and executables.
- *exp-results*, the collection of experiment results.

The user has to make sure that the following files/directories exist in the *dcx-source* directory:

- source files: *dcx.c*, *coalloc.c*, *scheduler.c*, *broker.c*, *resmonitor.c*, *thr_mgmt.c*, *list_mgmt.c*, *io_mgmt.c*.
- header files: *coalloc.h*, *scheduler.h*, *broker.h*, *resmonitor.h*, *thr_mgmt.h*, *list_mgmt.h*, *io_mgmt.h*.
- the *makefile* and its header file, *header_gcc32dbgpthr*.
- the *outputs* directory.
- Perl scripts used by DCS: *idle_procs.pl* and *cluster_load.pl*.
- Perl scripts for analyzing results: *getresult* and *getsurveillance*.

To compile the source, the user should enter the *dcx-source* directory and type one of the following commands:

```
$ make
$ make dcs
```

If everything is fine, the user will get an executable, called *dcx*, which is the command to submit job requests. The user can also clean up all files besides those in the *dcx-source* directory, by entering the command:

```
$ make clean
```

A.2 Preparing user applications and job requests

Before a user submits job requests, he must be sure that job requests have been specified correctly. Ordered job requests are stored in *.rsl* files, while unordered job requests in *.uno* files. Actually, unordered jobs is also specified in RSL, but they lack the *resourceManagerContact* parameters. Several ordered and unordered jobs can be submitted at once in a *.bat* file (batch file). So, a batch file keeps a list of ordered or unordered jobs. The location of each job request file must be specified in full format, without characters like '~', '.', and '..'.

The executables of the applications must be located on the same host as the *dcs* command, and the user must record the location of the applications in the *executable* parameter in each subjob. The users does not have to copy the executables to other clusters. However, if the executables require input files, users must copy the input files to every destination cluster and record their locations in the *directory* parameters. If the executables do not need any input file, the users can omit the *directory* parameter altogether, but if the parameters are still used, the directories must be truly located in the corresponding clusters.

The *poisson-case* and *cpi-globus-case* directories contain job requests as well as source codes for the corresponding user applications. The users should check whether the executable of an application has been created. If not, the user must compile its source code. For example, the user can compile the source code of the Poisson application with the command:

```
$ make poissonT_globus
```

A.3 Submitting job requests through *dcs* command

Everytime a user wants to run the *dcs* command, he must be sure that his proxy is still valid. To check whether or not it is valid, he can enter:

```
$ grid-proxy-info
```

and see the *timeleft* attribute.

If the proxy is expired, he must activate it again by entering the command:

```
$ grid-proxy-init
```

The system will ask the user to enter his username and password.

Users can submit a single or multiple job requests through DCS according to the following command:

```
$ ./dcs [[-fcfs | -fpfs] | [-bestfit | -worstfit]] [reqfile1 | reqfile2 ...| reqfileN]
```

The default option for the scheduling policy is FCFS, and for the fitting algorithm it is worst-fit. If the options are not specified, they will be automatically be employed by DCS. The following are some examples of entering commands to submit job requests through DCS.

If the user wants to submit an unordered job using the default options, he should enter:

```
$ ./dcs ../poisson-case/poisson_4_32.uno
```

This example shows the user can submit job requests located in any directory as long as the right location of the job requests is entered. Unlike the location of request files in a batch file, the location of job requests here can be specified by using characters '~', '.', and '..'.

If the user wants to submit several job requests using FPFS and worst-fit, he can enter one of these commands:

```
$ ./dcs -fpfs poisson_4_32.uno poisson_4_16.uno poisson_4_64.rsl  
$ ./dcs -worstfit -fpfs poisson_4_32.uno poisson_4_16.uno poisson_4_64.rsl
```

The user can also submit multiple job requests in a batch file as follows (using FPFS and best-fit):

```
$ ./dcs -fpfs -bestfit poisson_4_32.bat
```

or in combination with other single job requests or other batch files (using FPFS and worst-fit):

```
$ ./dcs -fpfs poisson_4_32_uno.bat poisson_4_32_mix-ord-uno.bat poisson_4_16.uno
```

A.4 Specifying which clusters are available/involved

If a user wants to specify which clusters will be available/involved in a DCS run, unfortunately he cannot do it by just tuning the *dcx* command. For now, users can only add or remove a line to or from the *idle_procs.pl* file, a Perl script that retrieves the number of idle processors. The line position will be highlighted by a comment, so users can locate it easily.

For instance, if a user wants to retrieve information from all clusters except from fs1 and fs2, the following line must be added to the script:

```
next if (($i==1)||($i==2));
```

So, if the user want to prevent more clusters to be involved, he must add those clusters in the line. Vice versa, if the user wants to involve all DAS clusters, he must remove the line. Adding or removing the line can be done through any text editor.

A.5 Analyzing the results

A DCS run will result in several files that are located in the *outputs* directory:

- the *result.log* and *surveillance.log* files.
- the *.out* files, each of which records the standard output text from the user executable.
- the *.err* files, each of which records the standard error text from the user executable.

The *outputs* directory is important! It must be located in the same directory where the *dcx* command is located. If it is not there, the user must create it.

If a DCS run represents an experiment, the user is recommended to rename the *outputs* directory with another informative name, but he must assure to re-create the *outputs* directory before he starts the next experiment. The renaming is necessary to put the results of an experiment into one directory to ease the user analyzing the results.

To analyze the results of an experiment, the user must extract data from the log files by running the *getresult* and *getsurveillance* Perl scripts.

The *getresult* script is made to extract data from *result.log*, which contains the statistics of every job submitted in this experiment (i.e., job ID, arrival time, response time, total run time, and net run time). For example, if the user wants to extract data from *result.log* located in the *uno-4x8-2avail-fcfs* directory, he can enter one of these commands:

```
$ perl getresult ../exp-results/uno-4x8-2avail-fcfs/result.log
$ ./getresult ../exp-results/uno-4x8-2avail-fcfs/surveillance.log
```

The *getsurveillance* script is made to extract data from *surveillance.log*, which are the total load (number of busy processors caused by all users) and DCS load (number of busy processors caused by DCS) in the involved clusters during this experiment. For example, if the user wants to extract data from *surveillance.log* located in the *uno-4x8-2avail-fcfs* directory, he can enter one of these commands:

```
$ perl getsurveillance ../exp-results/uno-4x8-2avail-fcfs/surveillance.log fs0 fs3
$ ./getsurveillance ../exp-results/uno-4x8-2avail-fcfs/surveillance.log fs0 fs3
```

This example shows that *getsurveillance* needs parameters to specify which clusters are involved in this experiment. If the user wants to know the system load from all clusters, he should give all the cluster IDs to the command, for example:

```
$ ./getsurveillance ../exp-results/uno-4x8-2avail-fcfs/surveillance.log fs0 fs1 fs2 fs3 fs4
```

The data extracted from the log files are stored in tab-delimited files so they can be processed directly by Microsoft Excel or MatLab to obtain the information the user wants, such as statistic parameters (average, standard deviation, etc.) or diagrams.

Appendix B

An Example of Unordered Job Request

Here is an example of unordered job request. It is specified in RSL but it lacks the *resourceManagerContact* parameters.

```
+
( &(count=8)
  (jobtype=mpi)
  (label="sub1")
  (environment=(GLOBUS_DUROC_SUBJOB_INDEX 0)
    (LD_LIBRARY_PATH /usr/local/globus/globus-2.2.4/lib/))
  (arguments= "8" "4")
  (maxWallTime=10)
  (directory="/home1/jsinaga/poisson")
  (executable="/home1/jsinaga/poisson/poissonT_globus")
)
( &(count=8)
  (jobtype=mpi)
  (label="sub2")
  (environment=(GLOBUS_DUROC_SUBJOB_INDEX 1)
    (LD_LIBRARY_PATH /usr/local/globus/globus-2.2.4/lib/))
  (arguments= "8" "4")
  (maxWallTime=10)
  (directory="/home1/jsinaga/poisson")
  (executable="/home1/jsinaga/poisson/poissonT_globus")
)
( &(count=8)
  (jobtype=mpi)
  (label="sub3")
  (environment=(GLOBUS_DUROC_SUBJOB_INDEX 2)
    (LD_LIBRARY_PATH /usr/local/globus/globus-2.2.4/lib/))
  (arguments= "8" "4")
  (maxWallTime=10)
  (directory="/home1/jsinaga/poisson")
  (executable="/home1/jsinaga/poisson/poissonT_globus")
)
( &(count=8)
  (jobtype=mpi)
  (label="sub4")
  (environment=(GLOBUS_DUROC_SUBJOB_INDEX 3)
    (LD_LIBRARY_PATH /usr/local/globus/globus-2.2.4/lib/))
  (arguments= "8" "4")
  (maxWallTime=10)
  (directory="/home1/jsinaga/poisson")
  (executable="/home1/jsinaga/poisson/poissonT_globus")
)
```

Bibliography

- [1] S. Banen, A.I.D. Bucur, and D.H.J. Epema. A measurement-based simulation study of processor co-allocation in multicluster systems. In Dror G. Feitelson, Larry Rudolph, and Uwe Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, volume 2862 of *Lect. Notes Comput. Sci.*, pages 105–128. Springer Verlag, 2003.
- [2] A.I.D. Bucur and D.H.J. Epema. The influence of the structure and sizes of jobs on the performance of co-allocation. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 1911 of *Lect. Notes Comput. Sci.*, pages 154–173. Springer Verlag, 2000.
- [3] A.I.D. Bucur and D.H.J. Epema. The influence of communication on the performance of co-allocation. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 2221 of *Lect. Notes Comput. Sci.*, pages 66–86. Springer Verlag, 2001.
- [4] A.I.D. Bucur and D.H.J. Epema. Local versus global schedulers with processor co-allocation in multicluster systems. In Dror G. Feitelson, Larry Rudolph, and Uwe Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, volume 2537 of *Lect. Notes Comput. Sci.*, pages 205–228. Springer Verlag, 2002.
- [5] Karl Czajkowski, Ian Foster, Nick Karonis, Stuart Martin, Warren Smith, and Steven Tuecke. A resource management architecture for metacomputing systems. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 1459 of *Lect. Notes Comput. Sci.*, pages 62–82. Springer Verlag, 1998.
- [6] Karl Czajkowski, Ian Foster, and Carl Kesselman. Resource co-allocation in computational grids. In *The Eighth IEEE International Symposium on High Performance Distributed Computing (HPDC-8)*, pages 219–228, 1999.
- [7] C. Ernemann, V. Hamscher, U. Schwiegelshohn, A. Streit, and R. Yahyapour. On advantages of grid computing for parallel job scheduling. In *Cluster Computing and the GRID, 2nd IEEE/ACM International Symposium CCGRID2002*, pages 39–46, 2002.
- [8] Ian Foster, Carl Kesselman, and Steven Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *Lecture Notes in Computer Science*, 2150, 2001.
- [9] Jean-Yves Girard. *Globus 2.4 and C++ applications: Staging files for grid jobs using Globus GASS Server*. IBM, France, June 2003.
- [10] V. Hamscher, U. Schwiegelshohn, A. Streit, and R. Yahyapour. Evaluation of job-scheduling strategies for grid computing. In *Lecture Notes in Computer Science*, volume 1971, pages 191–202, 2000.
- [11] <http://www.cs.wisc.edu/condor/>. The condor project.
- [12] <http://www.globus.org/>. The globus alliance.

[13] <http://www.openpbs.org/>. Portable batch system.

[14] <http://www.platform.com/products/LSF/>. Platform load sharing facility.

[15] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice-Hall, Inc., 2nd edition, 2001.