



European Community Sixth Framework Programme

RESEARCH INFRASTRUCTURES
Integrated Infrastructure Initiative

DEISA DISTRIBUTED EUROPEAN INFRASTRUCTURE FOR SUPERCOMPUTING APPLICATIONS

CONTRACT NUMBER 508830

Resource management in the core DEISA infrastructure

Deliverable ID: DEISA-D-SA3-1B

Due date: April 30, 2005
Actual delivery date: May 15, 2005
Leading contractor for this deliverable: CINECA, Italy

Project start date: May 1st, 2004
Duration: 5 years

Dissemination status: PP

Document Keywords and Abstract

Keywords:	DEISA, HPC, Grid, LoadLeveler, job migration.
Abstract:	

Copyright notices

© 2004 DEISA Consortium. All rights reserved. This document is a project document of the DEISA project. All contents are reserved by default and may not be disclosed to third parties without the written consent of the DEISA partners, except as mandated by the European Commission contract 508830 for reviewing and dissemination purposes. All trademarks and other rights on third party products mentioned in this document are acknowledged as own by the respective holders.

Table of Content

Project and Deliverable Information Sheet.....	Erreur ! Signet non défini.	Supprimé : 2
Document Control Sheet.....	Erreur ! Signet non défini.	Supprimé : 2
Document Status Sheet.....	Erreur ! Signet non défini.	Supprimé : 3
Document Keywords and Abstract.....	2	Supprimé : 4
Table of Content.....	3	Supprimé : 5
List of Figures.....	4	Supprimé : 6
List of Tables.....	4	Supprimé : 6
1. Introduction.....	5	Supprimé : 6
1.1 Executive Summary.....	5	Supprimé : 7
1.2 Document structure.....	6	Supprimé : 7
1.3 References and Applicable Documents.....	6	Supprimé : 8
1.4 Document Amendment Procedure.....	6	Supprimé : 8
1.5 List of Acronyms and Abbreviations, Short Glossary.....	6	Supprimé : 8
2. The AIX supercluster.....	8	Supprimé : 8
2.1 Abstract perspective.....	8	Supprimé : 8
2.2 DEISA homogeneous infrastructure.....	9	Supprimé : 10
3. DEISA core infrastructure : scheduling layer architecture.....	13	Supprimé : 10
3.1 Conceptual architecture.....	13	Supprimé : 11
3.2 Set up the homogeneous infrastructure : general agreements.....	14	Supprimé : 15
4. DEISA core infrastructure: scheduling layer implementation.....	19	Supprimé : 15
4.1 Installation of LL 3.3.0.....	19	Supprimé : 16
4.2 Gap analysis.....	21	Supprimé : 21
4.3 First implementation.....	21	Supprimé : 21
4.4 Pending issues and final implementation.....	28	Supprimé : 21
4.5 Conclusions.....	29	Supprimé : 23
5. Appendix A : Description of the IBM LoadLeveler (architecture and functionalities).....	31	Supprimé : 23
		Supprimé : 30
		Supprimé : 31
		Supprimé : 33

List of Figures

Figure 1 – Basic reference stack	<u>9</u>	Supprimé : 11
Figure 2 - DEISA core site basic reference stack.....	<u>10</u>	Supprimé : 12
Figure 3 - DEISA core site extended reference stack.....	<u>11</u>	Supprimé : 13
Figure 4 - DEISA core enhanced reference stack.....	<u>11</u>	Supprimé : 13
Figure 5 - DEISA homogeneous infrastructure functional description	<u>12</u>	Supprimé : 14
Figure 6 – DEISA core infrastructure conceptual architecture.....	<u>13</u>	Supprimé : 15
Figure 7 - Examples of job submission.....	<u>22</u>	Supprimé : 24
Figure 8 - Flowchart of the SUBMIT_FILTER script, which moves DEISA jobs to <i>deisa</i> class.....	<u>23</u>	Supprimé : 25
Figure 9 - Flowchart of the JOB_REROUTING script which moves jobs out of the <i>deisa</i> class. This flowchart applies to each job in the <i>deisa</i> class.....	<u>25</u>	Supprimé : 27
Figure 10 - Final job submission implementation.....	<u>30</u>	Supprimé : 32
Figure 11 - job lifecycle part I.....	<u>34</u>	Supprimé : 36
Figure 12 - job lifecycle part II.....	<u>34</u>	Supprimé : 36
Figure 13 - Workflow in a LL-MC.....	<u>38</u>	Supprimé : 40

List of Tables

Table 1 - List of acronyms and glossary	<u>7</u>	Supprimé : 9
---	----------	--------------

1. Introduction

1.1 Executive Summary

The objectives for the period project month 7 to project month 12 were “(a) the deployment of the global scheduler software on the proof of concept sites. Tests and fine tuning needed to guarantee production quality of the platform, as the dedicated network becomes operational, and (b) the extension to the remaining core sites (EPCC and CSC).”

Due to some latencies in the delivery of the selected product for the global scheduler software (IBM LoadLeveler (LL)) the activity related to point (a) has been a little bit slower than forecasted, while point (b) is a clear refuse because CSC and EPCC are not core sites. CSC will install the new LL release in the near future (when the product has reached a more mature status) while EPCC cannot install it at all because the machine is owned and managed by the HPG-X consortium.

Nevertheless the SA3 has reached the following targets:

1. a conceptual architecture for the global scheduler software has been developed and it has been agreed: this architecture is valid for a heterogeneous infrastructure (so the homogeneous infrastructure is considered as special case). It clearly demonstrates part of the metascheduler framework and its functionalities;
2. the global scheduler software on the proof of concept sites has been installed and tested (on test machines, not on production ones) and it has been demonstrated that it is not yet sufficiently mature to be used on real production systems. The weakness found were reported to the software provider and there is a proven commitment from the software provider to support all the DEISA requirements and requests for update;
3. even though the product is not yet completely ready, the core sites agreed upon a subset of keywords that should be used by the DEISA users. Moreover the core sites agreed on some operational aspects raised by the introduction of the global scheduler software in the production environment.

1.2 Document structure

In section 2 the document presents the conceptual architecture and the practicalities of how it maps on DEISA. Section 3 addresses the steps needed to design the scheduling layer architecture. Section 4 presents the first implementation made with the actual LL capabilities. Conclusions are presented in the last section. The Appendix describes the IBM LL design and functionality and reports on the experience made in installing it.

1.3 References and Applicable Documents

- [1] DEISA Primer Guide
- [2] DEISA, SA3 deliverable, D-SA3-1A
- [3] System Design for LoadLeveler Multicenter Support-CS3C, version 2.6
- [4] Using and Administering LoadLeveler, version 3.2, Chapter 1

1.4 Document Amendment Procedure

People just talk to each other. Possibilities include : phone, email, letter.

1.5 List of Acronyms and Abbreviations, Short Glossary

Term / Acronym	Definition
API	Application Programming Interface: a set of library routine definitions with which third party software developers can write portable programs. Examples are the Berkeley Sockets for applications to transfer data over networks, those published by Microsoft for their Windows graphical user interface, and the Open/GL graphics library initiated by Silicon Graphics Inc. for displaying three dimensional rendered objects.
CC-NUMA	Cache Coherent Non-Uniform Memory Access
CLM	CLuster Metric
DTJ	DEISA Translated Job
DUJ	DEISA User Job
GPFS-MC	IBM proprietary Global Parallel File System Multi Cluster
GUI	Graphical User Interface
HPS	High Performance Switch
IA64	Intel Architecture 64 bit, the Itanium processor
IP	Internet Protocol

JCF (jcf)	Job Command File
JO	Job Object
LAN	Local Area Network
LL	LoadLeveler
LL-MC	LoadLeveler - Multi Cluster
LoadLeveler	IBM proprietary batch scheduler subsystem
LPAR	Logical Partition
Meta-scheduler	The batch system capability to manage/address distributed batch schedulers available on different machines/sites.
OS	Operating System
TCP	Transmission Control Protocol
UID	User Identifier
UNICORE	UNiform Interface to COmputational REsources
WAN	Wide Area Network

Table 1 - List of acronyms and glossary

2. The AIX supercluster

2.1 *Abstract perspective*

Using a general abstract perspective, a system can be summarized by the *hardware layer*, on top of which runs the *operating system* (OS) (which has several functions, as for example accounting) and a *communication layer* (as, for example, the Message Passing Interface library or the TCP/IP layer able to address both a WAN/LAN or an interconnection network between the processors). On top of this layer sits the *resource manager layer*, which exposes and handles the resources (hardware – such as processors, memory, interconnection network, disks – but also software – such as license tokens). Important services offered to the end-users by the resource manager are the ability to submit, monitor, query and delete a job. The resource manager can be considered as the “worker”, that piece of software that maps the work onto the resources as requested by the “director”. The so-called “director” is the *batch scheduler layer*.

The batch scheduler contains the policies on resource usage (e.g. the dedicated/non-dedicated usage of a set of processors, the queue mapping, the queue physical limits as for example a limited amount of memory and the time limit for jobs residing in the queues). The workload balance inside the system is performed via the batch scheduler. The batch scheduler can perform several services, batch scheduler accounting for example, can help in understanding better job accounting, as opposed to process accounting, which by itself results in only a collection of consumed CPU cycles that are not linked to any job identifier. Advance reservation, the capability to reserve a certain amount of resources for a predetermined amount of time starting from a given point in time, is, if implemented, a service of the batch scheduler. On top of the services offered in the batch scheduler layer it is possible to build new services for homogeneous and heterogeneous batch scheduling. The synthetic image of our reference stack is given in Figure 1.

Sometimes, especially in practical issues, the distinction between the resource manager and the batch scheduler is not evident, this happens mostly because the software product that implements the batch scheduler requires a proprietary resource manager, so the two tightly connected layers appears as a unique layer. For example the IBM LoadLeveler (LL) comes with the so-called ‘backfill scheduler’ and its own resource manager, the backfill scheduler uses the resource manager API to implement the policies. But, the backfill scheduler can be switched off and substituted with another one, for example the MAUI scheduler.

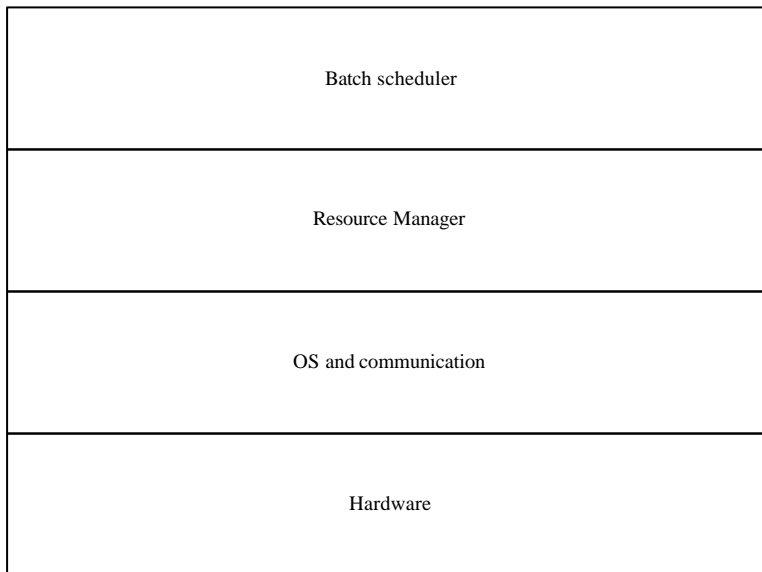


Figure 1 – Basic reference stack

2.2 DEISA homogeneous infrastructure

The DEISA core sites reference picture is given in Figure 2: the hardware is based on the IBM Power 4 processors and on the IBM pSeries High Performance Switch (HPS), the IBM AIX 5.2 operating system and the IBM LL batch suite composed of the resource manager and the backfill scheduler, run on this hardware.

CINECA, FZJ, IDRIS and RZG may differ in the number of processors, kind of processor (Power 4 versus Power 4+), clock speed, minor version of the operating system (or patch level) and LL but their reference stack can be easily represented as it has been done above, that is the reason why the four DEISA core sites can be really considered as homogeneous.

The initial phase involves four IBM Power4 platforms [1] :

- ? FZJ-Jülich (Germany): P690 (32 processor nodes) architecture, incorporating 1312 processors. Peak performance is 8.9 Teraflops.
- ? IDRIS-CNRS (France): Mixed P690 and P655+ (4 processor nodes) architecture, incorporating 1024 processors. Peak performance is 6.7 Teraflops.
- ? RZG-Garching (Germany): P690 architecture incorporating 896 processors. Peak performance is 4.6 Teraflops.

- ? CINECA (Italy): P690 architecture incorporating 512 processors. Peak performance is 2.6 Teraflops.

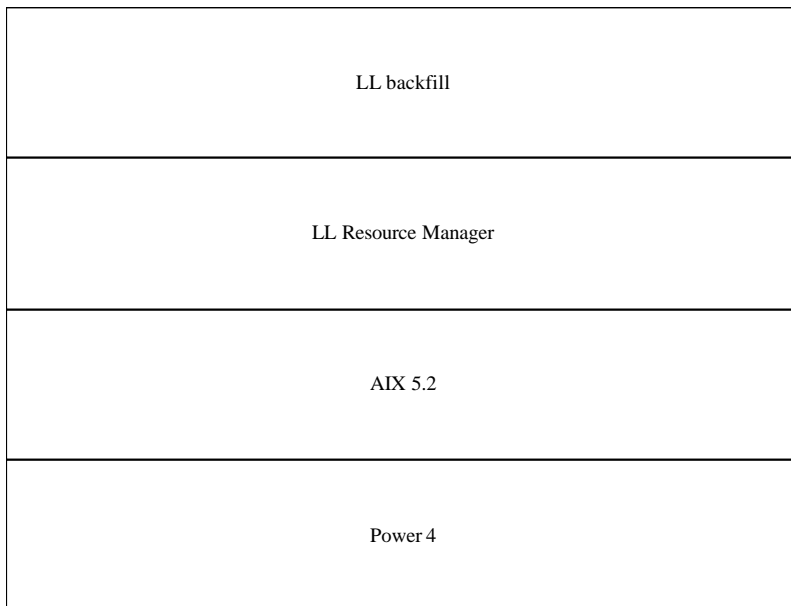


Figure 2 - DEISA core site basic reference stack

As mentioned earlier, the resource manager offers the ability to submit, monitor, query and delete a job. LL offers a graphical user interface (*xloadl*) able to simplify access to the batch scheduler facilities, but this GUI is suitable only for this particular batch system. The DEISA partners decided to address the problem of ease of access for users to the batch scheduler (so simplifying the access to the infrastructure) by adding on top of the batch scheduler layer an *access functionality* composed of a group of tools able to solve the problem for most of the batch scheduling systems. The access layer is composed of portals and the UNICORE middleware. Portals are GUIs that can hide batch scheduler complexities and syntax, while UNICORE also adds a powerful complex workflow manager able to handle chains of jobs spanning several machines. So the single core site reference picture can be expanded as in Figure 1.

At the start of the project, all DEISA core sites were running LL version 3.2., which did not offer the multi cluster capability, nor did it support the inter-site job rerouting feature, nor the advance reservation feature; for these reasons all these sites will have to upgrade to LL 3.3 which offers all these features. *Job rerouting* is another functionality that can be represented in the stack. The upgrade of LL means the complete substitution of the resource manager and of

the batch scheduler, it requires the development of inbound and outbound filters and a common rerouting policy agreement.

The DEISA core site reference stack will be so enhanced to form a homogeneous super cluster, as in Figure 4.

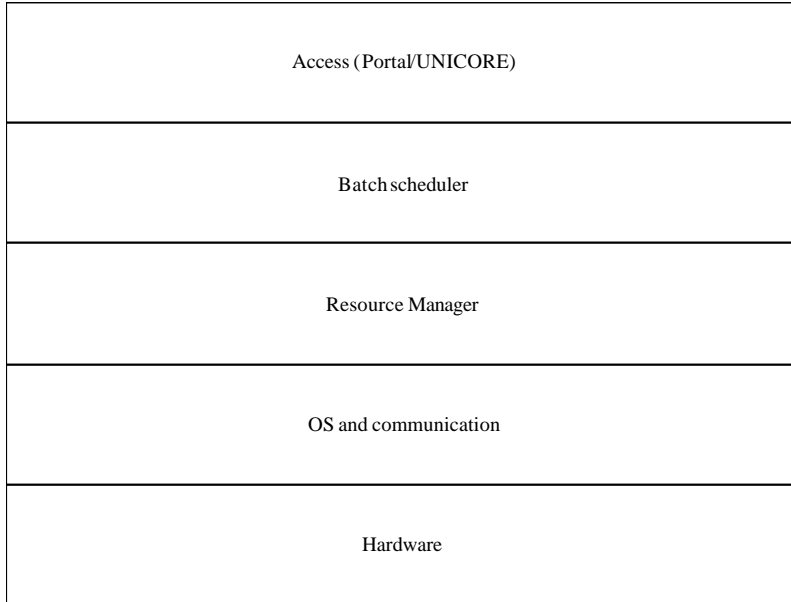


Figure 3 - DEISA core site extended reference stack

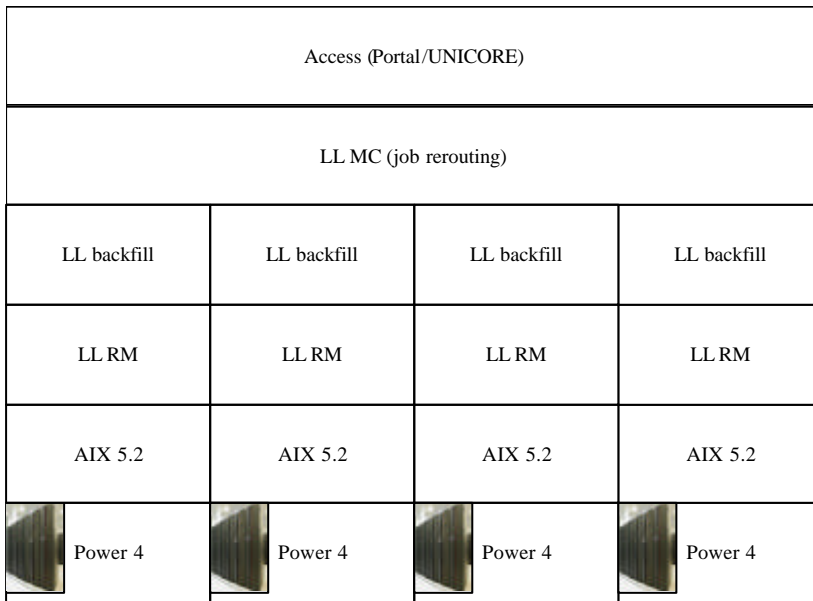


Figure 4 - DEISA core enhanced reference stack

The functionalities presently available can be summarized as in [Figure 5](#). It can be seen that in this figure the workflow management functionality (incorporated by UNICORE) has also been highlighted.

Mis en forme: Français
France
Supprimé : Figure 5

Access
Workflow management
Job rerouting
Policies implementation through the scheduler (workload,advance reservation, accounting)
Resource manager
OS and communication
Hardware

Figure 5 - DEISA homogeneous infrastructure functional description

Code de champ modifié
Supprimé : 5
Mis en forme: Français
France
Mis en forme: Français
France
Mis en forme: Français
France

3. DEISA core infrastructure : scheduling layer architecture

Initially a conceptual architecture has been designed that has been used as reference for the practical implementation. This task reported in [2], is briefly reviewed in section 3.1. Obviously the implementation is strictly dependent on existing site constraints and must rely on the product features. To respect the site constraints some general agreements were needed a priori, mainly on the keywords that would be allowed and those that would not. This homogenisation task is reported in section 3.2.

3.1 Conceptual architecture

DEISA core infrastructure conceptual architecture is summarized in Figure 6, and it has been introduced in [1].

Supprimé : Figure 6

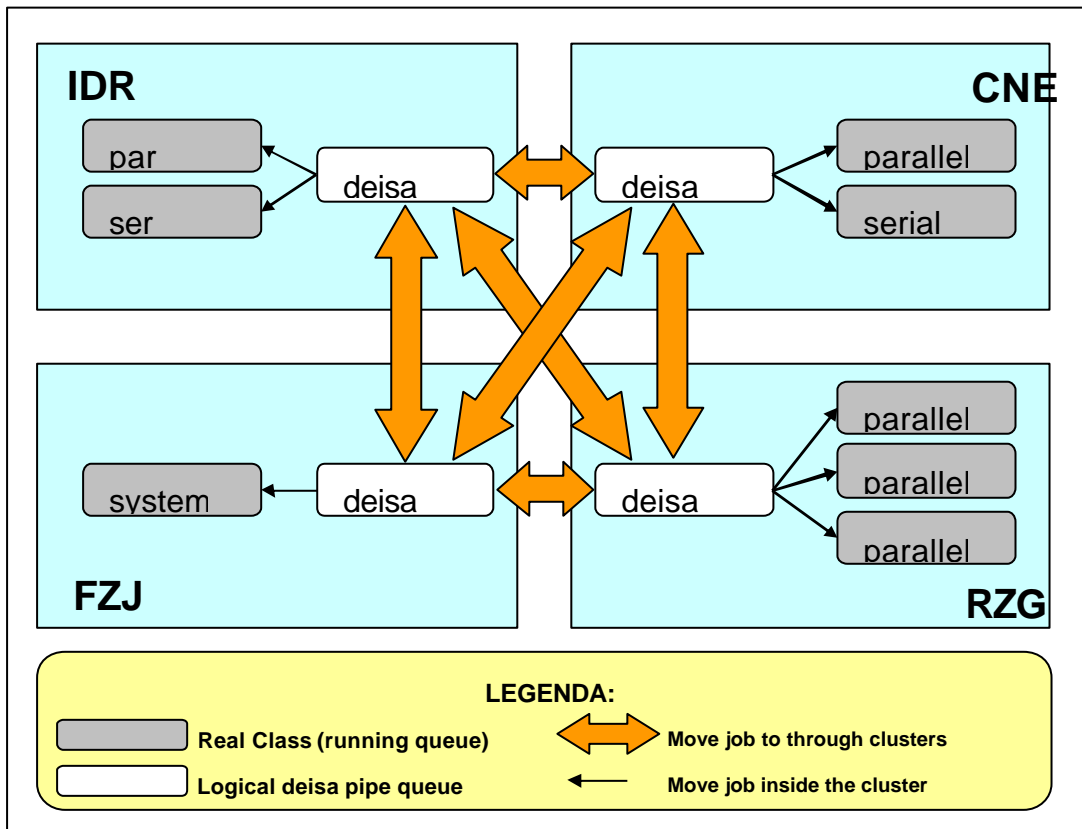


Figure 6 – DEISA core infrastructure conceptual architecture

For production purposes each site has already configured LL with a certain number of queues; DEISA will be not intrusive but will set up a routing mechanism by means of a pipe queue or some other technique; when the job is received, it can be piped to a site queue or it can be routed to a remote site. This mechanism will be implemented using new LL features.

3.2 Set up the homogeneous infrastructure : general agreements

General consideration concerning DEISA jobs

It is clear that not all jobs can be rerouted without change because of site-specific requirements, for this reason the DEISA partners must agree on a common subset of job specifications which will be correctly handled (and eventually satisfied) by the four core sites.

The following three major points need to be noted

- ? Jobs submitted as "DEISA" jobs cannot themselves submit other jobs, as this could result in undetermined behaviour.
- ? Today UIDs are static but in order to be compatible with future requirements, DEISA partners must not rely on a specific DEISA UID range when they set up the LL-MC.
- ? Jobs consisting of multi-job steps can exhibit undetermined behaviour in a distributed environment. DEISA will not allow this feature to be used initially by "DEISA" jobs, but will reconsider this decision in the future.

Acceptable job keywords

As previously stated, DEISA job specifications must respect some constraints in order to be run successfully. Specifically, some keyword values have to be specified while others must not be specified (or, if they are, they can be ignored or modified). In the following section such keywords will be examined. From a practical point of view, in order to ameliorate the impact of these constraints on the user's normal working practices, LL

filters will modify appropriately some job attributes in order to comply with the agreed rules. The following definitions will be used in the description:

- ? a DEISA User Job (DUJ) - the script written by an end user;
- ? a DEISA Translated Job (DTJ) - the script filtered and modified which is the output from the `llsubmit_filter`, that is used by LL to create the DEISA Job Object;
- ? a DEISA user job command file (JCF) – an exact copy of just the LL keywords present in the DUJ;
- ? a DEISA Job Spool – either the executable, if the job specified the “#@executable” LL keyword, otherwise an exact copy of the DUJ, including the LL keywords and the rest of the job script provided by the user;
- ? a DEISA Job Object (JO) - a LL entity containing all the job attributes.

Keywords for LL-MC

In order to be a LL-MC job, the DTJ must contain the “*cluster_id*” keyword. Considering the fact that some sites could also internally set up a LL-MC, it has been preferred to set up a special keyword to identify DEISA LL-MC jobs.

The “*requirement*” keyword with Feature == “DEISA” has been chosen to identify DEISA jobs, i.e.

@requirement = (Feature == “DEISA”)

The “*requirement*” keyword is mandatory within the DUJ, and sufficient for future identification.

The `llsubmit` argument “-X” allowing the user to specify a cluster from the command line must not be specified.

The “*cluster_id*” keyword must not be specified by the user. The “*cluster_id*” keyword has to be set up within the DTJ by the filter in case of immediate routing and the choice could be “*any*” or a list of clusters.

Keywords for MPI parallel jobs

The “*total_tasks*” keyword has been chosen as common keyword to manage task assignment. As already discussed before, it has to be set up by the filter to generate a DTJ in the case where a site allows other keywords to their users. This keyword is

supported by the backfill and gang schedulers, not by the basic scheduler, though this should not affect any of the DEISA sites.

Considering the fact there are different types of architecture (two at present) where the number of CPUs in a node differs from one site to another, the only keyword which could work on all of the architectures with "total_tasks" is the "blocking" keyword. This keyword has to be set up if the user omits it, and the best set up is:

@ blocking=unlimited

For other task assignment keywords, two choices are available to a DEISA site:

- ? deny the keyword and reject the job;
- ? accept the keyword and translate it into *total_tasks* and *blocking* keywords (this is the preferred option).

The "not_shared" keyword is not allowed in any kind of keyword to generate a DTJ and considering that for parallel job, a "network" keyword has to be set up (possibly by the user), otherwise - for MPI jobs - the submit filter will insert as default or to modify the DUJ keyword to:

@ network.MPI_LAPI=sn_all,shared,US

Keyword consideration for OpenMP parallel jobs

The "resources" keyword with a specification of "ConsumableCpus" keyword should be allowed to manage multi threads jobs.

Memory keywords

The "resources" keyword with a specification of "ConsumableMemory" keyword is allowed, to manage memory requirements for those sites running IBM's Workload Manager subsystem.

The "data_limit" and "stack_limit" keywords are not considered mandatory within the DUJ, but must be set up after submit filtering process in the DTJ.

Time keywords consideration

Because of differences in computer architectures, the specification of CPU and wallclock time is a problem. A job that takes 60 minutes to run on a 1.9 GHz POWER4 system may take nearly twice as long on a 1.3 GHz system and half as long on a POWER5 system. If the user specifies the "cpu_limit" and/or the "wall_clock_limit", the value the user provides should be the value that is consistent with the architecture at his/her site. This value will be normalised when it is inserted in the DESA token in the DTJ. Then if the job is run locally nothing needs to be done. While, if the job is routed to another cluster, the receiving site has to substitute the time value in the LL keyword, that it calculates using the normalised value and an understanding of that site's computer architecture.

Class keyword

It is recommended not to specify the "class" keyword in the DUJ.

If the site chooses to run the job locally without trying to send it immediately to another site, it could insert a production class name, during the submit filtering. Otherwise it will be up to the remote site to determine how to modify the class value to a real remote site queue.

If the site chooses to send the job somewhere else during *llsubmit* command execution, it must add

@ class = DEISA

or replace the DUJ class name with "DEISA".

Requirement keyword

The "requirement" keyword is used to define a DEISA job with the "(Feature == "DEISA")" requirement. It is recommended not to specify other requirements of this keyword in a DUJ. If DEISA partners accept them inside a DTJ, it would be the task of the inbound filter to:

- ? erase them (it would not reflect the initial demand of the user)
- ? or modify them, if this can sensibly be done;
- ? or reject the routed job (it could cause unpredictable behaviour).

Resources keyword

Apart from the "ConsumableCpus" and ConsumableMemory resources, DEISA partners can accept other resource options and either modify them, erase them or reject the routed job.

Checkpoint keyword

Some sites could need to checkpoint jobs. In order to achieve such a process, the keyword

@ checkpoint = yes

would have to be inserted. If necessary, this could be inserted by the receiving site through the inbound filter.

Execute permission consideration

A problem occurs during the execution of the job, if the submitted script does not have execute permission; the job fails. DEISA partners have to take care about this feature and find an agreement concerning the process which will verify that the script has the correct access or which will set up the correct permissions.

Pathname consideration

During *llsubmit* process, LL registers the absolute PATH of the current working directory at the time of submission. This PATH is used if the user specified the job's output or input by a relative PATH. If the absolute PATH does not exist on the execution node, LL sometimes creates the PATH or the job might remain indefinitely in "HOLD" status if the LL keyword "ACTION_ON_MAX_REJECT" is set to "HOLD". DEISA partners have to pay particular attention to choose the same mount point on each of the core sites for the GPFS-MC.

For any other keywords, the receiving site can do one of three things: accept the keyword, refuse the job or overwrite the keyword

4. DEISA core infrastructure: scheduling layer implementation

As stated at the beginning of section 3, it is important to have in mind an architecture but, obviously, the implementation is strictly dependent also on the product features. In this section the historical evolution of the LL-MC DEISA experience is presented: installation, the gap analysis performed, a first implementation that has been useful to give further inputs to IBM, and a second (and as of now final) implementation that is going to be deployed when the new product release will become available. Clearly the reference architecture developed is biased (the logical routing queue is not physically implemented as a class), nevertheless the original conceptual value remains true: a job can be executed remotely if it is put in a logical routing queue.

4.1 Installation of LL 3.3.0

Platform

The LoadLeveler Multi-Cluster (LL-MC) version between the four core-sites (IDRIS, RZG, CINECA and FZJ) was installed in a test environment which was very close to the future production environment. The test systems were already connected in the dedicated wide area DEISA network.

Separate test systems had to be provided, installed and configured at each core site in order not to interfere with the production systems. The same nodes which were already used for GPFS-MC installation a couple of months before were used for LL install.

IDRIS had three p655 nodes, two equipped with 4 CPUs and one with 8 CPUs dedicated to the DEISA environment. These machines were connected internally with HPS. Externally all these machines were connected to the DEISA-network with 1Gigabit ethernet interface.

RZG had a full p690 node divided into four LPARs having 8 CPUs each dedicated to the DEISA test-environment. The four machines were connected internally and externally with the DEISA-network via 1 Gigabit ethernet interface.

FZJ had four p690 LPARs consisting of 2 CPUs each dedicated to the DEISA test-environment. The four machines were connected internally and externally with the DEISA-network via 1 Gigabit ethernet interface.

CINECA had four p690 LPARs with 8 CPUs each dedicated to the DEISA test-environment. The four machines were connected internally with HPS and externally with the DEISA-network via 1 Gigabit ethernet interface.

Software

LL 3.3.0 beta release required at least one node on each site providing a cluster. These nodes had to be connected to the DEISA-network and able to communicate all to all. The version of AIX had to be 5.2 ML 03 or higher. Furthermore, it was recommended to have the same OS version on each site. The LL Software was required to be at the same level on each site.

Installation and testing

In the end of January 2005, specialists from the four core sites and ECMWF met at IDRIS for the first installation process with IBM. During a session lasting two days, a first install of the beta release of LL-MC between the 4 core sites was achieved.

First tests allowed us to verify the new LL-MC functionalities. DEISA partners were able to execute remote execution and able to reroute previously queued jobs in a site towards another site with the new *llmovejob* command. A secured configuration using SSL was installed. The new copy file feature and new LL environment variables were tested. A thorough analysis of the CLUSTER_METRIC and CLUSTER_REMOTE_JOB_FILTER filters started.

A detailed list of installation problems and incorrectly functioning features was given to IBM. A report containing design change requirements was initiated in order to be able to get the maximum flexibility with the remotely submitted or rerouted jobs.

4.2 Gap analysis

The current version of LL-MC does not contain some of the features which are necessary to implement fully the architectural design depicted in [Figure 6](#). The main problem is that the remote filter used to modify an arriving job (namely the CLUSTER_REMOTE_JOB_FILTER) is not a real filter, but a user-exit. It is not possible for this user-exit to change other than a small fraction of the job's configuration. In effect it can only accept or reject the job as it is. Such a limitation prevents us from fully implementing the architectural design using only an appropriate LL-MC configuration and features (CLUSTER_METRIC, CLUSTER_REMOTE_JOB_FILTER, etc.).

Supprimé : Figure 6

For this reason, DESIA partners decided to design and develop a first implementation of the DEISA infrastructure, described in [4.3](#), which takes advantage not only of the LL-MC features, but also of some ad hoc scripts.

Supprimé : 4.3

4.3 First implementation

Even lacking the necessary LL features to achieve a full implementation of the DEISA infrastructure as described in [Figure 6](#), DEISA partners tried to develop an incomplete but working routing system by means of some work-around solutions. This effort was made in order to gain familiarisation with the new architecture.

Supprimé : Figure 6

A job is considered to be a DEISA job only if it requests the DEISA feature, as stated in [3.2](#). Only DEISA jobs are considered from now on.

Supprimé : 3.2

The implementation follows a two-phase approach: during the first phase (submission), all the DEISA jobs are automatically queued into a local class named *deisa*; during the second phase (re-routing and execution), jobs are moved out of this class, and are assigned to a local or remote cluster to execute.

Phase one – job submission

As described above, when a DEISA job is submitted, the CLUSTER_METRIC script is evaluated to establish the most suitable cluster to execute the job, based on the value of the *cluster_list* keyword, and on dynamical constraints. In our first implementation, the

CLUSTER_METRIC script simply returns the name of the local cluster, independently of the *cluster_list* parameter.

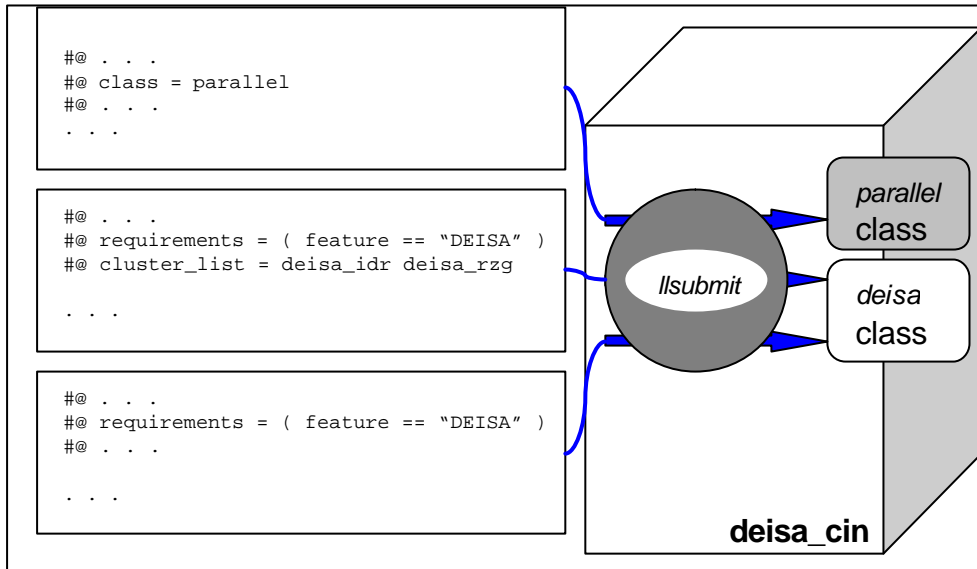


Figure 7 - Examples of job submission.

In order to allow the job to be scheduled, it is necessary to assign a class to it; so, the filter adds "*class = deisa*" to the job configuration parameter. This *deisa* class, which is defined on every DEISA cluster, does not have starters (i.e. is a pipe queue). Following this ideally the job must be modified to adapt to the DEISA requirements: site specific LL keywords values must be translated to allowed DEISA values and have to be saved in a LL keyword, to be retrieved during the CLUSTER_REMOTE_JOB_FILTER execution to enable it to adapt the job to the remote site. Unfortunately with the LL features available in March, the LL class was the only value that could be modified.

In summary: each DEISA job is submitted onto a *deisa* class on the local cluster, independently of the *cluster_list*. If the job does not specify a *cluster_list*, it is assumed to be *any*. If the job does specify a *cluster_list*, it is substituted with *any*. Since the *deisa* class has no starters, the job will wait endlessly in this class.

Figure 7 is an example of three different job submissions.

The flowchart of the SUBMIT_FILTER is shown in Figure 8.

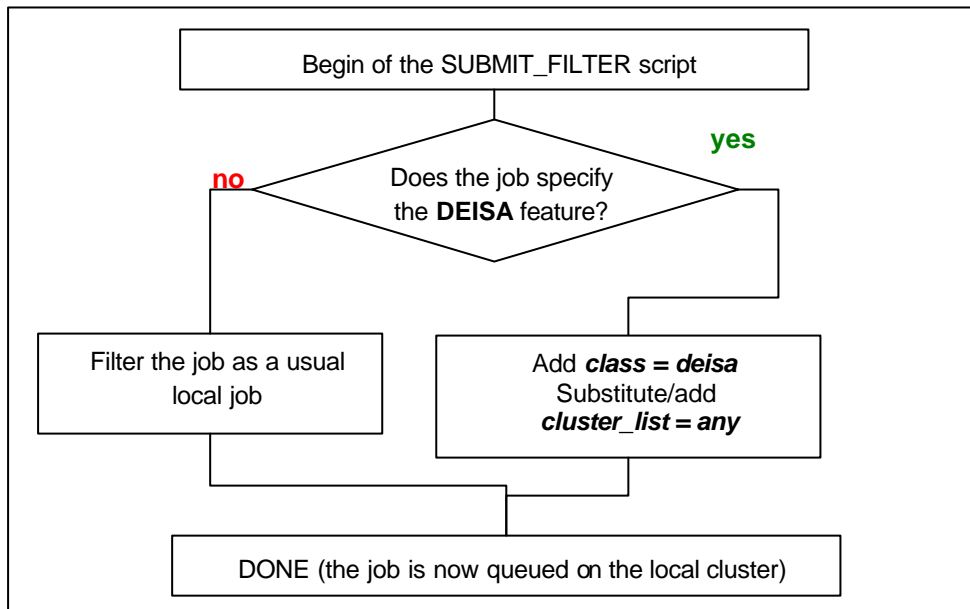


Figure 8 - Flowchart of the SUBMIT_FILTER script, which moves DEISA jobs to *deisa* class.

Phase two – job rerouting and execution

After submission all the DEISA jobs are left in Idle state in the *deisa* class on the local site. A script named JOB_REROUTING (see Figure 9) is run every minute via *crond*. This script parses every job waiting in the local *deisa* class, and assigns each of them to the local cluster or to a remote cluster for execution. Specifically, for each idle job in the *deisa* class, the script gets the *cluster_list*, sorts it with respect to a metric, and tries to “move” it onto the first cluster of the sorted *cluster_list*:

- ? if this cluster is the local one, the job is moved simply by changing the *class* parameter (and eventually some other parameters): the new class is an executing class, that is a class with starters. The JOB_REROUTING script must establish the value for this class parameter based on the job resource requests, and on the local policies.
- ? if the target cluster is a remote one, the job is moved by executing a *llmovejob* command. This command could fail (for example if the *initialdir* does not exist on the target cluster, or the CLUSTER_REMOTE_JOB_FILTER does not accept the

job) and, in this cases, the job is moved onto the second cluster in the sorted *cluster_list*, and so on.

Notice that the entire *cluster_list* could be parsed without finding any valid target cluster; in this case, the job should be left in the *deisa* class (it will be parsed again later, when maybe it will be able to run on some cluster). Note also that, in general, the job could be refused also by the local cluster for execution. For the sake of simplicity, Figure 9 does not consider these possibilities.

When a job is moved to a remote cluster, the class parameter is left unchanged; so, it will be parsed again by the script onto the newly designated executing cluster, which can decide to move it again.

Clearly this implementation allows that the same job reaches the same executing cluster several times: it could happen that dynamic constraints prevent a job from running on a cluster for a while, but that afterwards this same cluster could be profitably exploited to execute the job. Nevertheless, it will be necessary to avoid that most time is spent in continuously bouncing the same job between the same two (or more) clusters. This could be easily avoided by the rerouting scripts, using the “*Schedd History*” parameter, which tracks each *schedd* that -from time to time- takes charge of the job. That could be done directly inside the cluster metric, lowering the score of the clusters which are already in the “*Schedd History*”. When a job has recently been moved to each valid cluster, without having found the possibility to run on it, it could eventually be left for a while in the local *deisa* class, without trying immediately a new rerouting.

Notice that the cluster metric used in Figure 9 is not exactly the same as the CLUSTER_METRIC script; the main difference is that CLUSTER_METRIC must provide only the name of the cluster to be used for execution, that is the first cluster of the sorted *cluster_list*; instead the cluster metric CLM used in the job rerouting script must provide the entire sorted list. Nevertheless, it is clear that the main part of the two scripts is exactly the same.

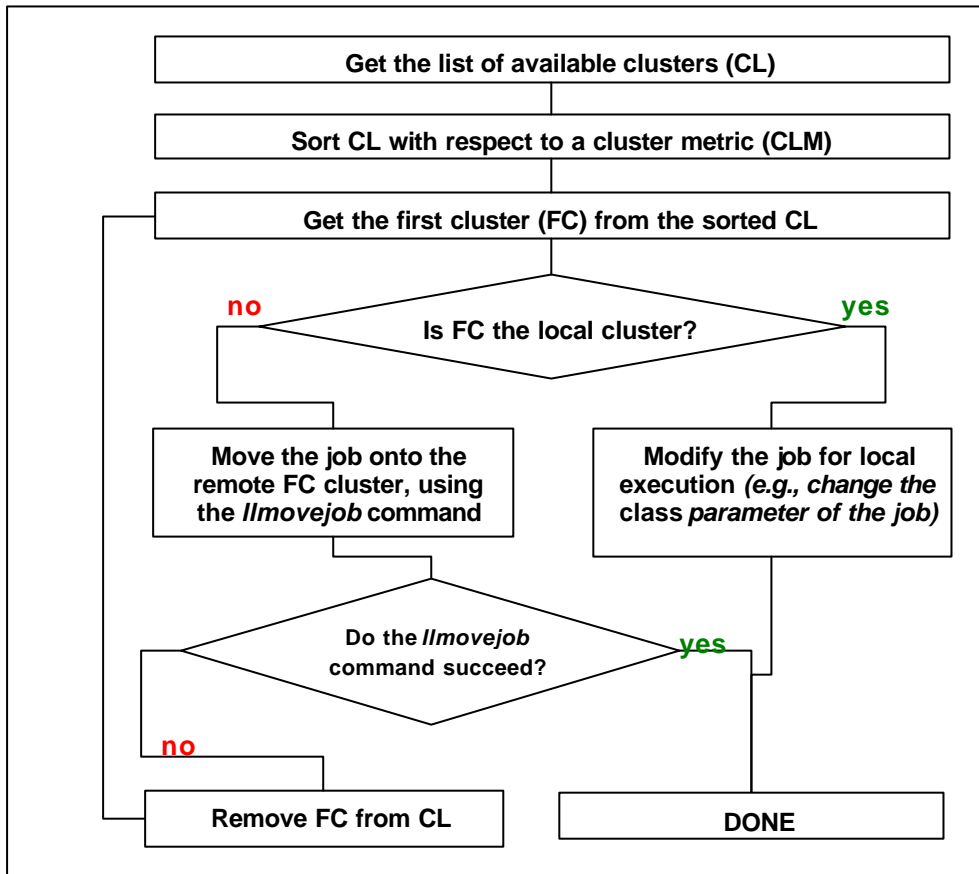


Figure 9 - Flowchart of the JOB_REROUTING script which moves jobs out of the *deisa* class. This flowchart applies to each job in the *deisa* class.

4.2.1 First experiences

From the user's point of view, the LL-MC features are essentially based on the ability to run some of the usual commands on remote clusters. A label is given to each cluster; this label can be used on the command line to specify the (remote) cluster on which the command has to be executed.

Example 1 shows how a user monitors his/her jobs on the cluster labelled "*deisa_fzj*" where *//q* is the usual LL command. Some of the commands can accept a special value

“all” after the `-X` flag, specifying that the command has to be executed on all the clusters.

```
> llq -u cne0cin3 -X deisa_fzj
===== Cluster deisa_idr =====
Id              Owner      Submitted  ST PRI Class      Running On
-----
deisa071.sp4.cine-.685.0 cne0cin3  3/22 10:48  I   50 deisa
```

Example 1

Example 2 shows the scheduled jobs for the user `cne0cin3` on all the clusters.

```
> llq -u cne0cin3 -X all
===== Cluster deisa_cin =====
llq: There is currently no job status to report.

===== Cluster deisa_fzj =====

Id              Owner      Submitted  ST PRI Class      Running On
-----
deisa071.sp4.cine-.685.0 cne0cin3  3/22 10:48  I   50 deisa

1 job step(s) in query, 1 waiting, 0 pending, 0 running, 0 held, 0 preempted

===== Cluster deisa_idr =====
llq: There is currently no job status to report.

===== Cluster deisa_rzg =====
llq: There is currently no job status to report.
```

Example 2

Example 3 shows that the `-X` flag accepts a clusters list:

```
> llq -X deisa_fzj deisa_idr
===== Cluster deisa_fzj =====

Id              Owner      Submitted  ST PRI Class      Running On
-----
j39d1m.zam.kfa-jue-.91.0 fzj302zm   3/3 14:49  I   50 system
deisa071.sp4.cine-.685.0 cne0cin3  3/22 10:48  I   50 deisa

2 job step(s) in queue, 2 waiting, 0 pending, 0 running, 0 held, 0 preempted

===== Cluster deisa_idr =====
llq: There is currently no job status to report.
```

Example 3

After submission, the job is scheduled on the cluster chosen by the `CLUSTER_METRIC`, which in the initial implementation is the local cluster. While the job is still in “IDLE” state, it can be moved onto a different cluster by using the `llmovejob` command; in our implementation this command is included in the `JOB_REROUTING` script.

As the job can be executed on a remote cluster, all the input files (needed for job execution) and all the output files (produced by the job execution) are stored in secure places from where they could be accessed later. Obviously, the user can put all the input/output data on a filesystem which is shared between all of the clusters, if such a filesystem exists (as in the case of DEISA); but this cannot clearly be a general solution¹, so, to this purpose, LL-MC provides two new keywords: *cluster_input_file* and *cluster_output_file* to enable files to be staged from the (local) submitting cluster to the (remote) executing cluster and vice versa. Both keywords require two paths: the first specifies the *local* path of the file (i.e., the path on the submitting cluster), while the second specifies the *remote* path of the file (i.e., the path on the executing cluster). These keywords can be used more than once, to specify more than one input and output file.

During submission, all the specified input files are staged (copied) to the scheduling host on the remote cluster that has been chosen to execute the job. If something fails during this copy, the submission itself fails. Notice also that the submission to the remote cluster fails if the *initialdir* path, which is the pathname of the current working directory in which the execution must start, does not exist on the remote cluster. For example, if a user does not specify the *initialdir* keyword, its value is the path of the (local) directory from which the *llsubmit* command is executed; but this directory may not exist on the remote cluster.

When the job is moved after submission, using the *llmovejob* command, the input files are also copied onto the new executing host.

After completion, all the specified output files are staged (copied) onto the local cluster.

LL-MC provides also two new variables *\$(home)* and *\$(user)*, which are expanded respectively to the home directory and username on the executing cluster; this allows the user to specify paths which are different on local and executing host, as reported in Example 4.

¹ Notice that, in general, the shared filesystem could have different mount points on different clusters.

```
# @ cluster_input_file = /sp4/userdeisa/cne0cin3/my/my.in, $(home)/my/my.in
# @ cluster_output_file = /sp4/userdeisa/cne0cin3/my/my.out, $(home)/my/my.out
# @ initialdir = /scratch/$(user)/my/run
```

Example 4

Clearly, $\$(home)$ could have different values on different clusters, even if the home directory is a shared filesystem.

4.4 Pending issues and final implementation

Since the initial installation, two updates of the product (in February 2005 and March 2005) have solved most of the major problems seen during the installation phase. Three problems are still outstanding.

IBM received with an open mind our change design requirements concerning the CLUSTER_REMOTE_JOB_FILTER and the need to be able to scan the file containing the user's LL keywords during the CLUSTER_REMOTE_JOB_FILTER execution. Two exchanges with developers in the IBM Labs have already contributed to a better definition of the major design modifications.

IBM is committed to addressing our architectural requirements during LL-MC development, so when all of the pending issues are solved, the scenario depicted in Figure 10 will be implemented. This figure shows an example with *llsubmit* of a job in clusterA and *llmovejob* in clusterB:

- the user submits a DEISA job containing the requirement "feature == "DEISA";
 - LL will store the jcf statements during *llsubmit* process in a separate *job.jcf* file in the spool directory; this means that LL will store in a file only LL Keywords (and the associated parameters) and nothing else (in particular, no DEISA token containing agreed DEISA keywords added during the submit filter in a script commented format that are meaningless for LL);
 - during the submit filter execution, a single LL keyword has to be used in order to store the generated "common DEISA user keyword values"; such a keyword could be the LL " #@comment " keyword (as in Figure 10). This field will be present in the JO inside LL. Obviously DEISA partners have to synchronise the format to be adopted to store our DEISA "token" in the "comment" keyword field.
- As a side effect, there is the benefit to keep the original user requirements saved

- separately from the site's modifications (sometimes though this is undesirable if the site chooses first to keep the job locally and afterwards chooses to move it);
- when a job is received by a remote site, a CLUSTER_REMOTE_JOB_FILTER script is executed. The first input to this script (passed as STDIN) is the previously created *job.jcf*, the second input is a link to the previously created *job.object* itself. Using the APIs, DEISA partners will be able to retrieve all the DEISA "token" contained in the "comment" keyword field, from which they will be able to get the necessary information to adapt the job to the site.

4.5 Conclusions

The testing performed by system administrators and potential users on the test machines has clearly demonstrated that LL-MC is not yet sufficiently mature for real production. Nevertheless the results of the functionality tests (with work-around solutions) have been very encouraging and the DEISA core sites have no doubts that once IBM solves the pending issues, this new feature of LoadLeveler's will enable us to progress with this part of the next phase of this project and as such the core sites are strictly committed to putting the global scheduling software into production at that time.

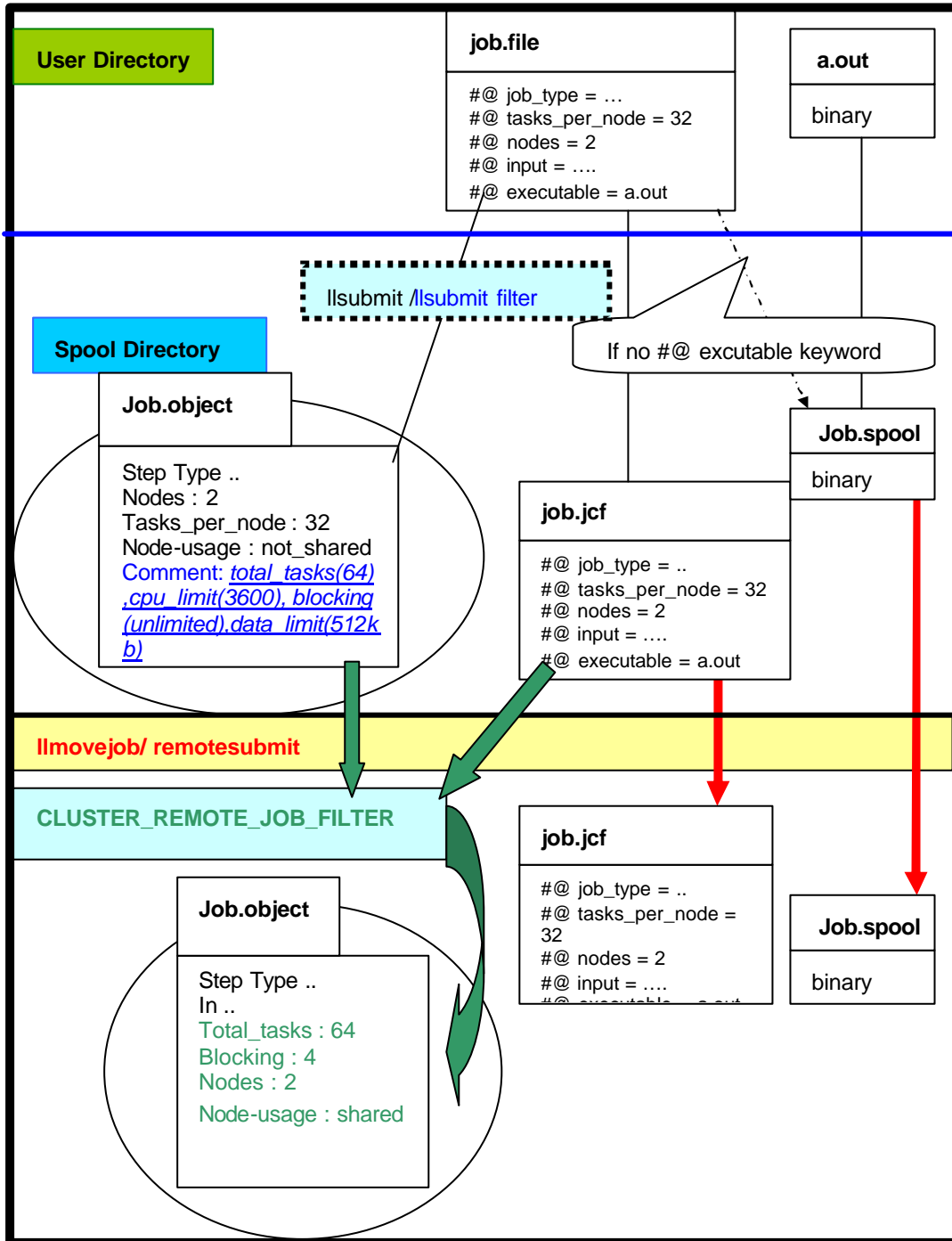


Figure 10 - Final job submission implementation

5. Appendix A : Description of the IBM LoadLeveler (architecture and functionalities)

LL is designed to manage both serial and parallel jobs over a cluster of servers [4]. Jobs are allocated to machines belonging to the cluster by a scheduler; the job allocation depends on the resource availability and some rule definitions made by the LL administrator. A user submits a job using a job command file (jcf), and the LL scheduler attempts to find resources within the cluster to satisfy the job requirements. At the same time, LL tries to optimise the cluster efficiency, maximizing the resources utilization and minimizing the job turnaround time.

To accomplish the above mentioned functionalities, the LL system is implemented as a set of daemons. It's an administrator issue to decide which daemons run on which machine of the cluster, not all of them need to run everywhere. The relevant ones are:

- ? **LoadL_master** : this daemon must run on all the machines in the cluster, except submit-only machines. This daemon manages all the other LL daemons on the local machine where it runs;
- ? **LoadL_schedd** : this daemon receives job submissions from *llsubmit* command and controls the jobs execution on the machines allocated by the *negotiator* daemon;
- ? **LoadL_startd** : this daemon maintains update information on the jobs state and resource usage on the local machine, and it sends these data to the *negotiator* daemon. It also spawns a starter process the scheduler daemon orders to begin a job on the local machine;
- ? **LoadL_starter** : this daemon runs a job and sends back status information on it to the *startd* daemon;
- ? **LoadL_negotiator** : this daemon runs only on the Central Manager machine. It collects status information from the *startd* daemons and, using those data, decides which job can be started on a pool of machines. This daemon is the only one which responds to the status request commands as *llq* and *llstatus*;
- ? **LoadL_GSmonitor** : this daemon runs on every machine of the cluster, and its function is to notify the *negotiator* daemon when a machine becomes unavailable;

- ? **LoadL_kbdd** : this daemon monitors mouse and keyboard activity on the local machine, and notify the *startddaemon* when some activity is detected.

Based on their LL role, the servers belonging to a cluster can be divided as follows:

- ? **Scheduling machine** : this is a machine where a *sched* daemon is running. When a user submits a job on a submitting machine, this job is placed in one of the scheduling machines queue; the scheduling machine contacts the central manager machine asking it to find one or more executing machines to run the job, and maintains an updated information on the job status;
- ? **Central Manager machine** : this is the machine where the *negotiator* daemon is running. It collects job status information from the scheduling machines and machine status information from the executing machines; finally the central manager machines contains all the rules to define the job execution on the whole cluster;
- ? **Executing machine** : this is a machine that can run a job (i.e. where the *start* daemon is running);
- ? **Submitting machine** (best known as **submit-only machine**) : this is a machine that can only submit, query or delete a job (through the *llsubmit*, *llq* and *llcancel* commands), but it's not able to run them.

The resources available in a LL cluster, and the desired run-time behaviour, are determined by the following configuration files:

- ? **LoadL_admin** : this file lists all the cluster machines, their type and network adapters, and all the classes (i.e. the various queues which defines the different possible resource allocations to jobs);
- ? **LoadL_config** : this file contains the default settings for the cluster, such as path names for log, spool and history files, daemon port numbers, configuration parameters for individual daemons, list of administrative users, scheduler type and associated operational parameters, parameters for ordering the job queues;
- ? **LoadL_config.local** : this file is a local configuration file that can override the global settings of *LoadL_config* for individual machines.

LL makes available many commands for users and administrators, which permits full access to LL functions. The most important ones are:

1. **llctl** : controls LL daemons (start, stop, reconfig, suspend) on all members of the cluster. This command can be issued only by administrators;
2. **llmodify**: changes the attributes or characteristics of a submitted job;
3. **llq**: returns information about job steps in the LL queues;
4. **llstatus** : returns status information about machines in the LL cluster;
5. **llsubmit** : submits a job to LL to be dispatched based upon job requirements in the jcf.

A lifecycle of a submitted job is:

1. A user submits a jcf through the *llsubmit* command on a submit machine.
2. This machine passes the job to a *LoadL_schedd* daemon running on a scheduling machine, which saves all the relevant job information on a local DB and puts the job in the local queue of submitted jobs.
3. The scheduling machine contacts the Central Manager machine to inform it that a job has been submitted.
4. The Central Manager machine, which knows the state of all the machines, searches for one or more machines (or nodes) that have available all the resources requested by the job.
5. When the resources have been found, the Central Manager authorizes the scheduling machine to start the job on a certain set of executing machines.
6. The scheduling machine contacts all the *start daemons* on each selected executing machine and tells them to execute the job.
7. The *start daemon* spawns a starter process which forks and executes the job.
8. When the user's job is completed, the starter process notifies the *start daemons*, and these notify the scheduling machine.
9. The scheduling machine examines the information from the *start daemons* and sends the final job status to the Central Manager machine.

Figure 11 and Figure 12 represent the above steps.

Mis en forme : Police :11 pt,
Police de script complexe :11

Supprimé : Figure 11

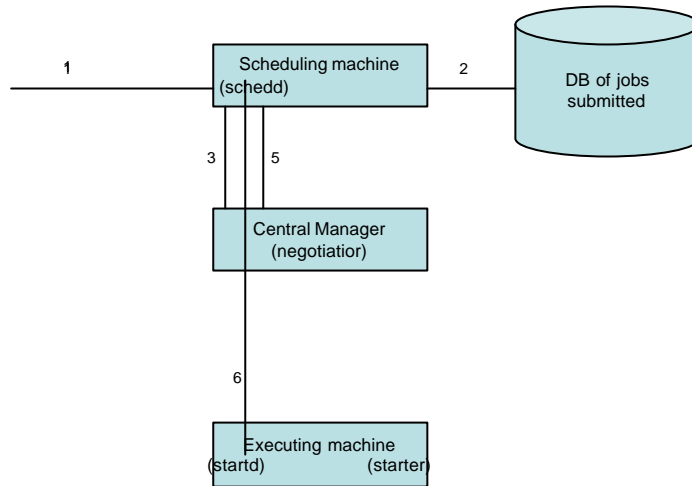


Figure 11 - job lifecycle part I

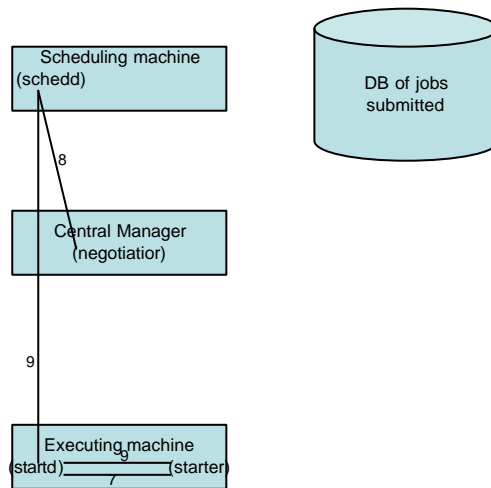


Figure 12 - job lifecycle part II

Recently, IBM has redesigned the LL architecture in order to provide multi cluster support. The main advantages of having multiple clusters are the scalability provided by keeping local clusters at a manageable size, and the ability of the users to access a greater amount of resources.

The following objectives were pursued:

- ? provide load distribution of jobs among multiple LL clusters at submit time;
- ? provide easy access to multiple LL cluster resources by users;
- ? provide control over multiple LL clusters by administrators;
- ? provide the manual transfer of an idle job from one cluster to another;
- ? provide for copying user input and output files between clusters;
- ? provide secure inter cluster communications.

To achieve the objectives, the following modification/addition to the LL architecture were made:

1. new administration file keywords were added to define the multi cluster environment. Every cluster now has to list all the other clusters belonging to the multi cluster and, for each of them, it has to know the hostnames of the machines which are used for inbound and outbound connections (i.e. the machines where the scheduler daemon is running);
2. in the jcf the user can specify a list of clusters where he/she wants the job to be executed (using the *cluster_list* keyword). The local scheduler daemon executes a user exit and decides (based on metric values from local and remote clusters) where the job has to be submitted;
3. the 'jobid' is still assigned by the local scheduler daemon, and is not modified when the job transfers from the submitting cluster to the executing cluster. On the other hand, the job status maintains information on all the clusters the job has reached, and they are available through the *llq* command;
4. each remote machine which has inbound/outbound connections with the local cluster must be previously authenticated as "trusted" machines;
5. UID mapping is supported for remote jobs (but not for root UIDs obviously); to achieve that, a user exit is executed on a cluster every time a job is submitted from another one;

6. there is a new command (*movejob*) that allows idle jobs to be transferred manually from one cluster to another;
7. if there is no shared file system between clusters, file transfers become necessary to send input files from the local (submitting) to the remote (executing) cluster and to retrieve the results from the remote to the local cluster. To achieve this staging of files, new jcf keywords (*cluster_input_file* and *cluster_output_file*) have been introduced to permit the user to specify which files have to be copied.

A job lifecycle in a multi cluster environment is as follows:

1. When a user wants to run a job on a remote cluster, he uses the *cluster_list* keyword to specify the list of desired clusters. The job is still submitted to the local scheduling machine, which assigns to it a jobid and selects the cluster where the job will be really executed.
2. The local scheduling machine transmits the job and the files to be staged to the local outbound gateway scheduling machine (which can obviously be the same server, but this is not compulsory). The local outbound gateway scheduling machine then transmits the job and the files to the remote inbound gateway scheduling machine.
3. Eventually, the remote inbound gateway scheduling machine performs UID mapping.
4. The remote inbound gateway scheduling machine executes all necessary submission checking on the job, and decides either to accept the job or to refuse it. If the job is rejected, the failure is returned to the local outbound gateway scheduling machine.
5. If the job is accepted, the local outbound gateway scheduling machine is informed of the successful transfer of the job and its input files, and the remote Central Manager is contacted for the "usual" job execution steps.
6. When the job completes, the remote inbound gateway scheduling machine informs the local outbound gateway scheduling machine and stages any output files.

Figure 13 shows the logical steps taken by a job from the submission to the end of its execution.

Let's focus a little bit further on step 1, in particular on the statement "selects the cluster where the job will be really executed": LL chooses a cluster from the *cluster_list* running the CLUSTER_METRIC script, provided by system administrators. The CLUSTER_METRIC sorts the *cluster_list* with respect to a metric that could be based on dynamical parameters, such as the workload of the clusters, and returns the name of the target executing cluster. When the local cluster tries to assign the job to the target executing cluster, the latter runs an exit (hopefully later a script filter) called CLUSTER_REMOTE_JOB_FILTER, which can accept or refuse the job and can change (almost) any of the job parameters, though currently only few job parameters are modifiable. If the job is accepted², the target cluster becomes the *designated executing cluster*, if not, the submission fails. On the *designated executing cluster* the process described can be repeated since LL-MC allows system administrators to move a job from a designated executing cluster to another while the job is in IDLE state, only when the job starts to execute does the currently designated executing cluster become the *effective executing cluster*.

² In order to allow the submission to successfully happen, it is necessary also that

- ? the target cluster can access the *initialdir*, which is the current working directory in which the execution starts, as specified by the LL keyword inside the jcf or set up as default by LoadLeveler to the directory from which the job has been submitted.
- ? all the input files could be successfully staged onto the target cluster.

In fact LL-MC allows users to specify input and output files inside the jcf, using appropriate keywords; the files are automatically staged in (out) before (after) the execution.

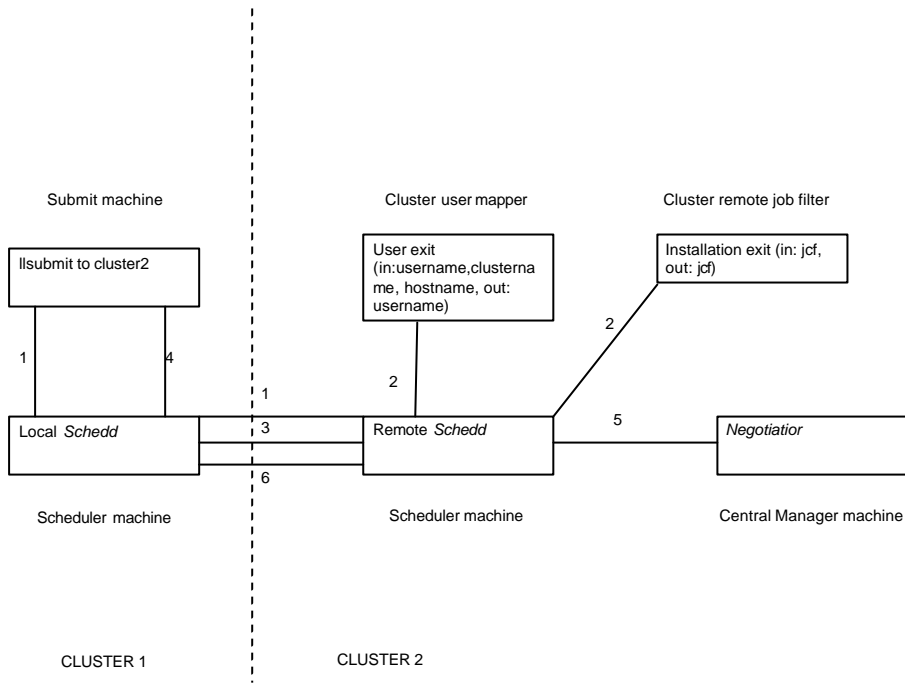


Figure 13 - Workflow in a LL-MC