

Report from visits to Condor (Madison) and Globus (ANL) teams

Massimo Sgaravatto

August 2, 2000

Release 1.2.3

Introduction

The goal is to implement a workload management system for productions.

These are scheduled activities, driven by the experiments and/or the physics groups, where the goal is to optimize the overall throughput.

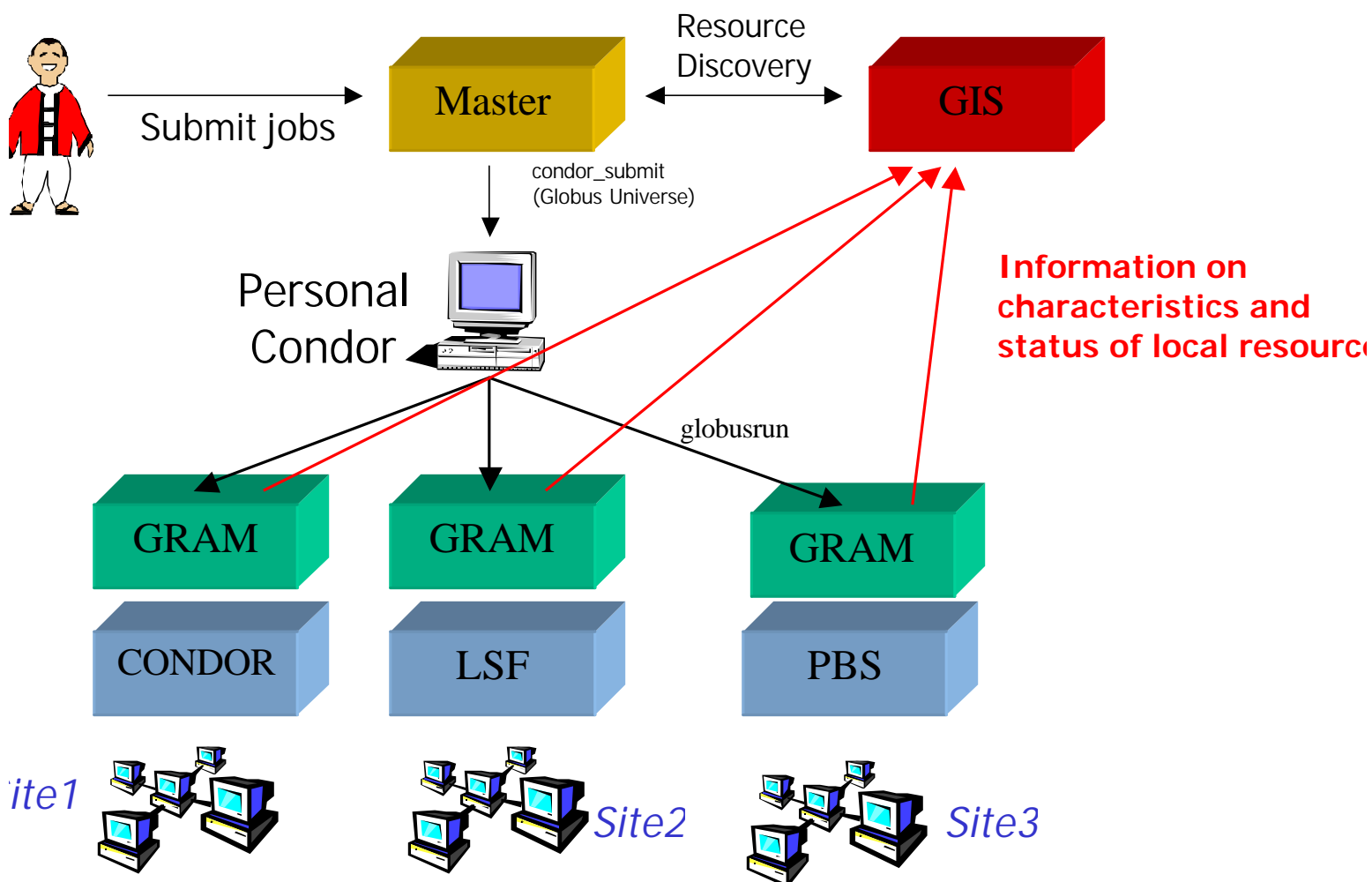
Two types of production activities can be identified:

- Monte Carlo productions
These are usually CPU bound applications, with a limited I/O rate, that usually just need as input small card files describing the characteristics of the events to be generated and geometry files describing the detector to be simulated, that can be staged in the executing machine without a significant cost.
Therefore for Monte Carlo production the goal is to build a system able to optimize just the usage of all available CPUs, maximizing the overall throughput. Code migration (moving the application where the processing will be performed) must be considered as implementation strategy.
- Data reconstruction and other production analysis
Reconstruction of real/simulated data is the process where raw data are processed, and ESD (Event Summary Data) are produced.
Production analysis includes Event Selection (samples of ESD of most interest to a specific physics group are selected) and physics object creation (the selected ESD data are processed, and the AOD, the Analysis Object Data, are created).
These are again scheduled activities, driven by the experiment and/or by physics groups that typically will be performed in the Tier 1 Regional Center possibly distributed in different sites.
For these kinds of applications the goal is again the maximization of throughput. Besides code migration, data migration (moving the data where the processing will be performed) must be considered as a possible implementation strategy: it could be profitable to move the data that must be processed to remote farms, able to provide largest CPU resources, but it must be

considered that the required data sets usually have a non negligible size, and therefore the cost for moving them must be taken into account.

In general it is necessary to consider “normal” jobs: we can’t assume, for example, that the executables can be relinked with the Condor library, in order to use features such as remote I/O and checkpointing.

The following picture describes the architecture that had been identified as a possible solution for the problem:



The computing resources (farms composed by commodity PCs) are managed by possible different local resource management systems (such as LSF, PBS, Condor, etc...).

It is possible to assume that there is a common shared file system between the machines in each site, but not between the machines in different sites.

The Globus GRAM service is used as a common interface to these different resource management systems, and the local GRAMs provide the GIS with information on characteristics and status of the local resources.

Personal Condor, with the Globus Universe mechanisms, is used to provide robustness and fault tolerance: even if there are problems in the submitting machine and/or in the executing machines and/or in the network, the user has the guarantee that the submitted jobs will be completed (if a job fails for example for a crash of the machine where it is running, it will be resubmitted).

The Globus Universe requires to explicitly define in which Globus resources the jobs must be executed. A master (that must be implemented) is the “smart” module of the system that decides where the jobs must be executed, considering the information published in the GIS.

Globus GRAM Service

To understand how the Globus GRAM service work, let’s consider an example where a user from a client machine submits a bunch of jobs (for example 100) to a remote Globus resource, configured as a front-end machine of a cluster managed by a local resource management system (for example PBS).

When the *globusrun* command is issued, in the front-end machine (the workstation running the Globus gatekeeper) of the PBS cluster, a process (the job manager) is created. The job manager is the process that actually submits the 100 jobs to PBS (the jobs are submitted all together), and keeps running until the jobs have been completed. When the jobs have been executed, the job manager exits.

Therefore in the front-end machine there is a running job manager for each *globusrun* command issued from the client (submitting) machines.

This can be a problem for scalability, because if many *globusrun* commands are issued, many job managers run in the front-end machines, and this could have a serious impact on the performance and robustness of this machine (but no tests have been performed to evaluate this problem).

When a user wants to control the status of the submitted jobs, with a *globus-job-status* command, the remote job manager is contacted. If it is running, it reports the status of the submitted jobs (ACTIVE, PENDING), while if the job manager doesn’t run anymore in the front-end machine, it is assumed that the jobs have been successfully completed.

In the current implementation the Globus job manager is not persistent. If it crashes, for example because the machine where it is running (the front-end machine) is rebooted, it doesn’t restart, while there could be jobs still running in the underlying resource management system (PBS, in our example).

Therefore there are “orphans” jobs without a job manager that “take care” of them. If the user who submitted the jobs performs a *globus-job-status* command, this reports, making a mistake, that the jobs have been completed.

So what is missing is a fault tolerant, robust, persistent job manager.

Moreover, Globus is not able to understand if a job has been successfully completed, or if it failed because of a problem of the executing machine.

Let's consider as example a job submitted to a remote Globus resource (for example a machine that simply uses the fork system call to run jobs), and for a crash of this machine, the job is lost. In this case *globus-job-status* simply answers that the job has been completed.

GRAM and LSF as underlying resource management system (solved and open problems)

- Using LSF as underlying resource management system for Globus, it is not possible to submit multiple "independent" instances of the same job.
For example the following command:

```
% globus-job-run lxde15.pd.infn.it/jobmanager-lsf -q mqueue -count 10 \  
-stdout -l /tmp/result -s /home/prg1 arg1 arg2
```

doesn't submit 10 instances of job *prog*: the option *-count* is translated in the option *-n* of the LSF *bsub* program (used to specify the number of processors required for a parallel job: *bsub* just allocates these processors, but the jobs are dispatched to a single machine: the first one). Therefore the result is not running 10 instances of the program in 10 different CPUs as requested (considering that the *mqueue* queue has been configured to run a single job for each CPU): the command just allocates 10 CPUs, but the jobs are dispatched just to a single one.

The problem could be solved running multiple instances of the *globusrun* command (in the previous example 10 times) but, as described above, this also means having multiple instances (10, in our example) of job managers running in the front-end machine, and this could be a problem if many jobs are submitted.

An other possible solution is modifying the *globus-script-lsf-** scripts (stored in the *<globus-deploy-dir>/libexec* directory after the deployment), in order to run the command *bsub* *x* times, if in the RSL expression the parameter *count=x* has been specified, having a single job manager for these multiple jobs.

- The parameter *freenodes* published in the GIS is not correct for a LSF cluster: it often reports a negative value.
This is because all the LSF scripts (stored in the *<globus-deploy-dir>/libexec* directory after the deployment) have been written considering an SMP shared memory machine, and not a cluster of PCs. Therefore these scripts must be modified.
- There was a problem related with the program *globus-job-status*: it reported that the jobs had been completed while they were still running.
To solve this problem it has been necessary to modify the file:

<globus-deploy-dir>/libexec/globus-script-lsf-submit

replacing the line:

```
job_id=`cat $LSF_JOB_OUT | ${awk} '{sub(/</, "", $2); sub(/>/, "", $2); print $2}'`
```

with:

```
job_id=`cat $LSF_JOB_ERR | ${awk} '{sub(/</, "", $2); sub(/>/, "", $2); print $2}'`
```

This is because LSF writes the id of the submitted job in the standard error instead of the standard output.

It is not clear if this is a problem with LSF 4.0 (the Globus LSF scripts have been previously tested by the Globus team considering a LSF 3.2 installation, and in this case the id of the submitted job is written in the standard output).

- The problem with the option *-publish-jobs* (used to publish in the GIS information related with the running Globus jobs) has been solved: it seems it is necessary to specify the parameter in two files:

<globus-deploy-dir>/etc/globus-services

and:

<globus-deploy-dir>/etc/grid-info-resource.conf

.

GRAM and Condor as underlying resource management system (solved and open problems)

- It is necessary to modify the file:

<globus-deploy-dir>/libexec/globus-script-condor-submit

of the front-end machine of the Condor pool, otherwise Globus assumes that for vanilla jobs the standard input/output/error must be redirect to /dev/null.

In the contrary in the file it must be written:

```
if [ "${condor_universe}" = "vanilla" ]; then  
  echo "Initialdir = ${grami_directory}"  
  echo "Input = ${grami_stdin}"  
  echo "Output = ${grami_stdout}"  
  echo "Error = ${grami_stderr}"
```

- There was a problem related with the program *globus-job-status*: as status it reported ACTIVE even if the jobs were pending.

The problem can be solved modifying the script:

```
<globus-deploy-dir>/libexec/globus-script-condor-poll
```

The line:

```
if [ 0 -lt `echo " R I " | grep -c " $val "` ]; then
```

must be replaced with:

```
if [ 0 -lt `echo " R " | grep -c " $val "` ]; then
```

and the line:

```
if [ 0 -lt `echo " U " | grep -c " $val "` ]; then
```

must be replaced with:

```
if [ 0 -lt `echo " U I " | grep -c " $val "` ]; then
```

- The problem with the option *-publish-jobs* (used to publish in the GIS information related with the running Globus jobs) has been solved: it seems it is necessary to specify the parameter in two files:

```
<deploy-dir>/etc/globus-services
```

and:

```
<deploy-dir>/etc/grid-info-resource.conf
```

However, if a cluster of jobs is submitted (specifying *count=x*, with $x > 1$), in the GIS the same id is used for all these jobs (the cluster id).

Other open problems related with the Globus GRAM service

- It is not possible to have the ids of the Globus jobs submitted in background (the ids returned when the commands *globus-job-submit* or *globusrun* with the option *-b* are used): the command *globusrun -l* doesn't work anymore.
The only possible solution is to query the GRIS/GISS of the Globus resource where the jobs have been submitted.

- The option `-publish-users`, that should publish in the GIS the content of the *grid-mapfile*, doesn't work properly.
The Globus team will analyze the problem.

Condor Globus Universe

The Globus Universe mechanisms have been tested, trying to submit Condor vanilla jobs on:

- a workstation running Globus, that uses the fork system call as job manager (case 1)
- a front-end machine (running Globus) of a LSF Cluster (case 2)
- a front-end machine (running Globus) of a local Condor pool (case 3)

Case 1

This is an example of condor submit file:

```
Universe = globus
Executable = /users/noi/sgaravat/CondorGlobus/ciao
GlobusRSL = (stdout=/users/noi/sgaravat/CondorGlobus/out)
log = /users/noi/sgaravat/CondorGlobus/log.$(Process)
Getenv = True
GlobusScheduler = lxde16.pd.infn.it/jobmanager-fork
queue 1
```

In this case the executable file must be stored in the file system of the executing machine and the output file is created in the file system of the executing machine (*lxde16.pd.infn.it*).

The submission of jobs (using `condor_submit`), the monitoring (using `condor_q`), the removing (using `condor_rm`) work fine.

When a job is removed using `condor_rm`, the process on the remote machine is immediately killed, but `condor_q` keeps reporting the job as running for a while (probably because the status of the job is checked by Personal Condor every 30 seconds).

Case 2

This is an example of condor submit file, that submits a job on queue *mqueue* of a LSF cluster, where *lxde15.pd.infn.it* runs Globus and has been configured as front-end machine:

```
Universe = globus
Executable = /users/noi/sgaravat/CondorGlobus/ciao
GlobusRSL = (stdout=/users/noi/sgaravat/CondorGlobus/out)(queue=mqueue)
log = /users/noi/sgaravat/CondorGlobus/log.$(Process)
Getenv = True
GlobusScheduler = lxde15.pd.infn.it/jobmanager-lsf
queue 1
```

The executable file must be stored in the file system of the executing machine and the output file is created in the file system of the executing machine as well.

The submission of jobs (using *condor_submit*), the monitoring (using *condor_q*), the removing (using *condor_rm*) work fine.

When a job is removed using *condor_rm*, the process on the remote machine is immediately killed, but *condor_q* keeps reporting the job as running for a while (likely because the status of the job is checked by Personal Condor every 30 seconds).

Case 3

This is an example of condor submit file:

```
Universe = globus
Executable = /users/noi/sgaravat/CondorGlobus/ciao
GlobusRSL = (stdout=/users/noi/sgaravat/CondorGlobus/out)
log = /users/noi/sgaravat/CondorGlobus/log.$(Process)
Getenv = True
GlobusScheduler = lxde14.pd.infn.it/jobmanager-condor
queue 1
```

The executable must exist in the file system of the executing machine, and the output file is created in the file system of the executing machine as well.

The submission of jobs (using *condor_submit*), the monitoring (using *condor_q*), the removing (using *condor_rm*) work fine.

Even in this case when a job is removed using *condor_rm*, the process on the remote machine is immediately killed, but *condor_q* keeps reporting the job as running for a while.

Problems with Globus Universe

- It is not possible to have the output file in the submitting machine. This is because in Globus it is not possible to have the output file in the submitting machine if the job has been submitted in background (it is not possible to use the options *-b* and *-s* together as parameters of the *globusrun* command).

The option *-s* of *globusrun* specifies that a Gass server must be activated in the client (Personal Condor, in this case), in order to be able to save the output file in the file system of the client machine. The Gass server is automatically shut down when the job has been completed, and therefore a connection between the submitting machine and the remote job manager is required. Using the option *-b* for the *globusrun* command, the job is executed in background, and therefore after the submission the connection between the client and the remote job manager is broken. This explains why the option *-b* and *-s* cannot be used together.

According the Globus team, a possible solution is to start a Gass server in the client machine (Personal Condor), that can then be used by possibly different multiple processes executed in remote Globus resources.

Therefore, after having started the Gass server, in the Condor submit file the output file could be specified as in following example:

```
.GlobusRSL = (stdout=https://lxde01.pd.infn.it:4560/./disk1/outfile)
```

where *https://lxde01.pd.infn.it:4560* is the id returned by the *globus-gass-server* command, run in the Personal Condor workstation.

So far no tests have been performed on scalability, and therefore it is not clear how many remote processes a Gass server is able to manage.

An other possible workaround is to copy back the output file (using for example *globus-rcp*, or *gsiftp*, or *scp*), but this can be done only when the job has been completed (without the possibility to check the output file from the submitting machine while the job is still running).

For what concerning the Gass service, according Tuecke and Martin this service will not be dismissed (as reported by other persons) since it is used by other Globus services (for example the Gram service).

- It is not possible to consider in the Condor submit file the parameters *output=* and *error=*. The workaround is to use the argument *GlobusRSL* in the condor submit file.

I.e:

```
GlobusRSL = (stdout=/home/sgaravat/out)(stderr=/home/sgaravat/err)
```

This problem will probably be solved in Condor 6.1.15 (but in any case the output and the error files will still be saved in the file system of the executing machine).

- If there are errors in the Condor submit file (for example the *output=* parameter has been specified, or it has been specified an executable that doesn't exist in the executing machine), the

condor_q command reports that the job is running, while the job is not actually running in the remote Globus resource.

It is not possible to understand about the problem checking the log file of the Condor job.

- The log file of the Condor job, making a mistake, reports as executing machine the submitting machine: therefore it is not possible to understand where the job has been actually executed.
- Even if the parameter:

Getenv=True

has been specified in the Condor submit file, all the variable definitions are lost for the job that must be executed in the remote resource (only some logical variables related with Globus are defined for the job in the executing machine).

In Globus there is not an equivalent RSL parameter, and according to the Globus team it will not be implemented, because it can be dangerous to “copy” all the environment variables from the submitting machine to the executing machine).

The Condor software should therefore use the RSL attribute *Environment* for each variable defined in the submitting environment.

- Sometimes the command *condor_submit* crashes when more than one jobs are submitted to a Globus resource
- The most important problem is that the current implementation of the Globus Universe is not able to provide robustness and fault tolerance.

Let's consider as example a configuration, where Personal Condor uses an LSF cluster as Globus resource.

Let's imagine that, with a single *condor_submit* command, a user submit 100 jobs to LSF, considering the Globus universe (for this example let's assume that there is not the problem previously described in submitting multiple instances of “independent” Globus jobs to an LSF cluster).

Personal Condor submits these 100 jobs to LSF all together. Then, it keeps polling (every 30 seconds) the remote job manager, checking the status of these jobs, using the Globus command *globus-job-status*.

As described before if, for a certain reason, the remote job manager crashes, but the jobs are still running on the LSF machines, the command *globus-job-status* answers that the jobs have been completed, and this is not true.

Moreover, if a job has been submitted to a remote Globus resource (for example that simply uses the fork system call to run jobs), and for a problem (for example a crash) on this machine, the job is lost, *globus-job-status* answers that the job has been completed.

Therefore Personal Condor has no way to understand if the job has been successfully completed, or if it failed for a problem in the remote resource (in this case the job should be resubmitted by Personal Condor).

Therefore there is fault tolerance only in the submitting machine side: if Personal Condor crashes, when it is up again it keeps checking the status of the jobs previously submitted to the remote Globus resource.

The Globus Universe architecture will deeply change with Condor 6.3 (that will be ready not before September 2000, being very optimistic).

In the new model, instead of considering a polling using *globus-job-status*, a persistent connection between Personal Condor and the remote job manager will be established, using the Globus APIs. Personal Condor will require an explicit notification when a job has been successfully completed. If it doesn't receive an explicit answer, and it is not able to contact the remote job manager anymore, it will assume that a problem occurred, and therefore it will have to manage the situation (for example checking that the job is not running, and then resubmitting it).

But this solution won't solve all the problems. For example, considering the previous configuration, what happens if a job has been submitted to LSF, and the machine where the job manager runs crashes, and then starts again ?

Since Globus doesn't save in a persistent way the status of the jobs, there will be an "orphan" job running on LSF (without a job manager that manages it), and the Globus client (and therefore Personal Condor) will not be able to "manage" it anymore.

To have a fault tolerance environment, Globus should be able to provide a persistent fault tolerant job manager.

An other possible solution could be a "direct interaction" between Personal Condor and the remote underlying resource management system (LSF, in our example), profiting of the exiting fault tolerant Condor architecture (this would require modifying the Condor *startd* software).

Condor GlideIn

If robustness and fault tolerance must be considered as fundamental requirements, that must be implemented in the workload management system, and it must be considered an environment where the executing machines are distributed in different sites, each one using a specific resource management system (LSF, Condor, PBS, ...), the only "ready to use" solution seems to be the Condor GlideIn architecture.

With GlideIn, what happens is that the Condor daemons (specifically, the *master* and the *startd*) are effectively run on Globus resources (machines that use the fork system call as job manager, or clusters managed by a resource managed system, with a workstation configured as Globus front-end machine). These resources then temporarily (the Condor daemons exit gracefully when no jobs run for a configurable period of time) become part of the considered Condor pool.

With this model, Personal Condor doesn't provide only robustness and fault tolerance, but also can act as a Master, since it decides in which resources the jobs must be executed, considering the Condor matchmaking system.

This solution could be viable for Monte Carlo productions, where more or less it is just necessary to find idle CPUs, while it could not be feasible (or it should be integrated) to manage other applications (such as reconstruction), where it is necessary to take into account also other parameters (i.e. the location of input files).

After having acquired a valid user proxy (using `grid-proxy-init`) the glidein procedure operates in two steps.

In the first one (when the option `--setuponly` is used), that must be considered only once, the Condor executables and config files are downloaded by a server in Wisconsin, using a particular ftp program developed by the Globus team. To perform this operation, the subject name of the user must have been defined in the *grid-mapfile* of the server in Wisconsin.

In the second phase (when the option `--runonly` is specified) the Condor daemons are executed in the remote Globus resource.

Various changes have been necessary to fix some problems:

- The `condor_glidein` script has been modified, in order to define in the glided machine the same value of the parameter `FILESYSTEM_DOMAIN` of Personal Condor (for Condor vanilla jobs it is necessary to have the same value of this parameter between the submitting and the executing machine).
It would be useful to have the possibility to define this parameter as option of the `condor_glidein` command.
- In the Condor configuration file of Personal Condor it has been necessary to modify the parameter `CLASSAD_LIFETIME` to 6 minutes (the default was 15 minutes).
- Since there are problems related with accounts defined in the NIS for machines running Linux RedHat 6.1, it has been necessary to use the dynamic version of `condor_starter` instead of the static version downloaded by the `condor_glidein` script. This required also installing in each executing machine a particular shared library (`/usr/lib/libbfd-2.9.1.0.23.so`).

The GlideIn mechanisms have been tested considering as Globus resource:

- a workstation that uses the fork system call as job manager (case 1)
- an LSF cluster, with a workstation configured as front-end machine (case 2)
- a local Condor pool, with a workstation configured as front-end machine (case 3)

Case 1

For the first phase of the GlideIn, the following command has been used:

```
condor_glidein --setuponly lxde16.pd.infn.it/jobmanager-fork
```

while for the second phase the following command has been considered:

```
condor_glidein --runfor 10 --runonly \  
"lxde16.pd.infn.it:2119/jobmanager-fork:/C=US/O=Globus/O=Istituto Nazionale di Fisica  
Nucleare/CN=lxde16.pd.infn.it"
```

The option *--runfor <mins>* defines that after *<mins>* minutes without running Condor jobs the remote Condor daemons exit gracefully. The default value is 20 minutes, while if the value 0 is specified the Condor daemons never exit.

The submission, the monitoring and the removing of Condor jobs (using *condor_submit*, *condor_q*, *condor_rm*) work fine.

It is also possible to explicitly “unghide” the machine (that therefore leaves the Condor pool), with a *condor_rm <id>* (*<id>* is returned when the *condor_glidein* script is executed), but it would be very useful to have a *condor_unghidein* command !

Case 2

For the setup phase of the glidein, the following command has been used:

```
condor_glidein --setuponly lxde15.pd.infn.it/jobmanager-lsf
```

For the second phase the following command has been considered:

```
condor_glidein --queue mqueue --runfor 10 -count 3 --runonly \  
"lxde15.pd.infn.it:2119/jobmanager-lsf:/C=US/O=Globus/O=Istituto Nazionale di Fisica  
Nucleare/CN=lxde15.pd.infn.it"
```

With this command it is asked that 3 nodes of the LSF cluster, associated to the LSF queue *mqueue*, will have to run the Condor daemons, and therefore they must join the Condor pool initially composed only by the Personal Condor machine.

However this doesn't happen, because, as explained above, the option *-count 3*, is translated in the RSL parameter (*count=3*), that is then translated in the LSF parameter *-n 3* of the LSF *bsub* program (that just allocates 3 processors, but the jobs are dispatched to a single one).

Therefore the result is not running 3 instances of the Condor master in 3 different machines as requested.

So for LSF the GlideIn procedure works fine only if a single host is glided in, or if the LSF pool is composed by a single machine with multiple CPUs.

The workaround could be running multiple instances of *condor_glidein*, considering the option *--count 1* each time. But in this case multiple instances of job managers (one per each glided in machine) would run in the front-end workstation, and this could be a problem if many machines must be glided in. In extreme cases, a hundred of Globus job managers could run in the front-end machine, querying the Condor scheduler every 30 seconds, which would really bog the machine down.

For this purpose the Condor team is going to modify the *globus-script-condor-** scripts, to have the queries cache the information on all Globus submitted jobs, and have the poll script use the cached queue results: this increases the latency of discovering job status changes, but also reduces the load of the job managers.

An other possible work around could be modifying the *globus-script-lsf** scripts, as described above, in order to be able to run multiple instances of the LSF *bsub* command, with a single job manager.

After having glided in a LSF machine, the submission, the monitoring and the removing of jobs (using *condor_submit*, *condor_q*, *condor_rm*) work fine, while it is not possible to explicitly “unglide” the machine with a *condor_rm <id>* (*<id>* is returned when the *condor_glidein* script is executed).

This is because the *condor_rm* is translated into the LSF command *bkill*, that sends a SIGKILL to the job (while a SIGTERM or SIGQUIT should be considered).

Probably this problem can be fixed working on the configuration files of LSF.

Case 3

For the first phase of the glidein, the following command has been used:

```
condor_glidein --setuponly lxde14.pd.infn.it/jobmanager-condor
```

while for the second phase the following command has been considered:

```
condor_glidein --queue mqueue --runfor 10 --runonly --count 3 \  
"lxde14.pd.infn.it:2119/jobmanager-condor:/C=US/O=Globus/O=Istituto Nazionale di Fisica  
Nucleare/CN=lxde14.pd.infn.it"
```

If all the machines of the local condor pool must join Personal Condor, the value for the parameter *--count* should be any number equal (or greater) to the number of machines composing the pool.

The submission, the monitoring and the removing of Condor jobs (using *condor_submit*, *condor_q*, *condor_rm*) work fine, but it is not possible to explicitly “unglide” the Condor pool, with a *condor_rm <id>*. This is because *condor_rm* sends a SIGKILL to the job (while a SIGTERM or SIGQUIT should be considered): this problem should be fixed in a future release of Condor.

The workaround could be doing *condor_off --master* for each machine that must leave the Condor pool, but this operation can not be performed from Personal Condor, because the name of the machine

running Personal Condor is not included in the parameter *HOSTALLOW_ADMINISTRATOR* of the machines that have been glided (to fix this problem the *condor_glidein* script must be modified).

Condor flocking

While the GlideIn mechanisms can be useful to “acquire” single Globus machines that use the fork system call as job manager, or clusters running for example LSF or PBS, this doesn’t seem a proper architecture to “acquire” an existing Condor pool. Since there are no reasons to go through Globus (unless the GSI service capabilities are required), in this case the Condor flocking architecture seems to be the most appropriate solution.

To evaluate these mechanisms, a flocking between the machine running Personal Condor and an existing local Condor pool has been implemented.

For this purpose it has been necessary just to define in the Condor configuration file of the machine running Personal Condor the following parameters:

```
FLOCK_HOSTS = lxde14.pd.infn.it  
HOSTALLOW_NEGOTIATOR_SCHEDD =  $\$(NEGOTIATOR_HOST)$ ,  $\$(FLOCK_HOSTS)$ 
```

(*lxde14.pd.infn.it* is the IP name of the central manager of the Condor pool), while in the machines of the Condor pool it has been necessary to include the IP name of the machine running Personal Condor in the *HOSTALLOW_READ* and *HOSTALLOW_WRITE* parameters.

The submission, the monitoring and the removing of jobs have been tested: no problems have been found.

I/O issues for vanilla jobs (for Condor flocking and GlideIn)

Considering an environment where in each site there is a shared file system between the “local” workstations, but these file systems are not shared between different sites and between the submitting machine (Personal Condor), there are some issues related with I/O that must be taken into account, when the Glide in and/or the Condor flocking mechanisms are used, and the considered jobs are vanilla Condor jobs.

For example if the *output=* parameter has been defined in the Condor submit file, the output file is created by Condor in the file system of the submitting machine; when the job runs on a “remote” machine, it is not able to find this file, and therefore the job crashes. This happens even if the specified pathname of the output file is “valid” also in the executing machine (that is a pathname that refers to a directory that exists in the executing machine).

There are two possible workarounds:

- Instead of submitting an executable file, a script file (that must exist just in the file system of the submitting machine) must be specified. This script actually runs the executable (that must be stored in the file system of the executing machine), and redirects the standard input/output/error to files in the file system of the executing machine.
Then the output files can be copied in the file system of the submitting machine (using for example *globus-rcp*, or *gsiftp*, or *scp*), but this can be done only when the job has been completed (without the possibility to check the output file from the submitting machine while the job is still running).
- The use of the *bypass* mechanisms (see below) that, using remote I/O, make possible to consider the standard input file in the submitting machine, and the standard output and standard error files created in the file system of the submitting machine as well.

Bypass

The *bypass* mechanisms make possible to redirect the standard input/output/error of a program to a remote machine.

Many of the ideas and techniques used in *bypass* were inspired by similar features in Condor, but *bypass* does not require Condor: they are separate programs.

Bypass requires that the program be dynamically linked (on some systems, you can use the *ldd* program to determine if a program is dynamically linked).

The *bypass* architecture is composed by two elements:

- the shadow
- the agent

The shadow is a program that runs in the machine where the I/O must be performed. A single shadow can be considered for multiple jobs running on possible different machines, and can be multithreaded (a thread is created for each remote process, otherwise a process is created for each job).

The following example shows how to start the shadow:

```
io_shadow -port 50000 -multithread
```

The agent is a shared library that must exist in the executing machine.

In order to consider the standard input, output, error in the remote machine, the following variables must be defined before invoking the job:

```
LD_PRELOAD <pathname of the shared library that implements the shadow>  
BYPASS_SHADOW_HOST <ip-address of the machine where the shadow is running>  
BYPASS_SHADOW_PORT <port> (the port associated to the shadow: 50000 in the previous example)
```

BYPASS_STDOUT <pathname of the standard output in the machine where the shadow is running>
BYPASS_STDIN <pathname of the standard input in the machine where the shadow is running>
BYPASS_STDERR <pathname of the standard error in the machine where the shadow is running>

For example, to use the bypass mechanisms for Condor vanilla jobs, a Condor submit file like the following one must be considered:

```
Universe = vanilla
Executable = /home/sgaravat/CMt/PythiaExeDir/mc.exe
initialdir = /tmp
Environment=LD_PRELOAD=/Bypass/io_agent.so;\
  BYPASS_SHADOW_HOST=193.205.157.15;\
  BYPASS_SHADOW_PORT=50001;\
  BYPASS_STDOUT=/tmp/TestIo/out.$(Process);\
  BYPASS_STDIN=/tmp/TestIo/in.$(Process);\
  BYPASS_STDERR=/tmp/TestIo/err.$(Process)
arguments= $(Process)
Getenv = True
queue 10
```

Bypass is a “new” software, still in a development phase, and some problems must be fixed, such as the security (in the current implementation any one could use a remote shadow, if the IP address and the port number of the machine where it is running is known), the support for shell scripts that redirect the standard input/output/error, the dependence on other shared libraries, etc...

Moreover, other tests to evaluate the capabilities and the robustness of this software are necessary.

In any case these activities are very promising and could be considered as a viable solution for our problems. For example, using the bypass mechanisms, the user who submitted the job could check the output/error files while the job is running, and therefore he could control how the job is working (errors, work done until that moment, etc...).

Conclusions

There are various problems that must be fixed in the Globus Universe mechanisms.

In particular the current implementation is not able to provide robustness and fault tolerance, and therefore at the moment it is not too much useful to consider Personal Condor with the Globus Universe mechanisms in the workload management system architecture.

It seems that the Globus team is going to redesign and reimplement the Globus GRAM service, in order to have a scalable, fault tolerant job manager, but this cannot be done in a short period.

If fault tolerance, reliability and robustness are considered important requirements that must be implemented, the only “ready to use” solution at the moment seems to be the GlideIn architecture and/or the Condor flocking mechanisms.

While GlideIn seems working with Globus machines that use the fork system call as job manager, and with Condor pools (but in this case the use of Condor flocking seems to be the most appropriate solutions) there is a problem that must be solved, related with the glidein of a LSF cluster (the problem should be solved simply modifying some Globus scripts).

Since vanilla Condor jobs must be considered, and in this case Condor requires a shared file system between the submitting machine and the executing machines (and we can't assume this), it is necessary to properly manage the I/O of the jobs.

Two possible solutions have been identified.

In the first one a shell script is submitted, so the input and output files must exist/are created in the executing machine.

The second solution considers the use of the bypass architecture, that, using remote I/O, allows to consider the input and output files in the file system of the submitting machine (Personal Condor). The bypass mechanisms are still in a development phase (problems such as security must be managed), but these activities are very promising and could be considered a viable and feasible solution for our needs.

Using the GlideIn and/or flocking mechanisms, Personal Condor acts also as a Master, since it decides, using the Condor matchmaking mechanisms, in which resources the jobs must be executed.

This can be considered a viable solution for Monte Carlo productions, where more or less the problem is just to find idle CPUs, while it must be investigated if it is feasible and viable to integrate and/or modify this architecture to manage also other production activities, such as data reconstruction, where other parameters (i.e. the location of input files, the available storage, etc...) must be taken into account.