Job Manager GridMD Component

1 Definitions

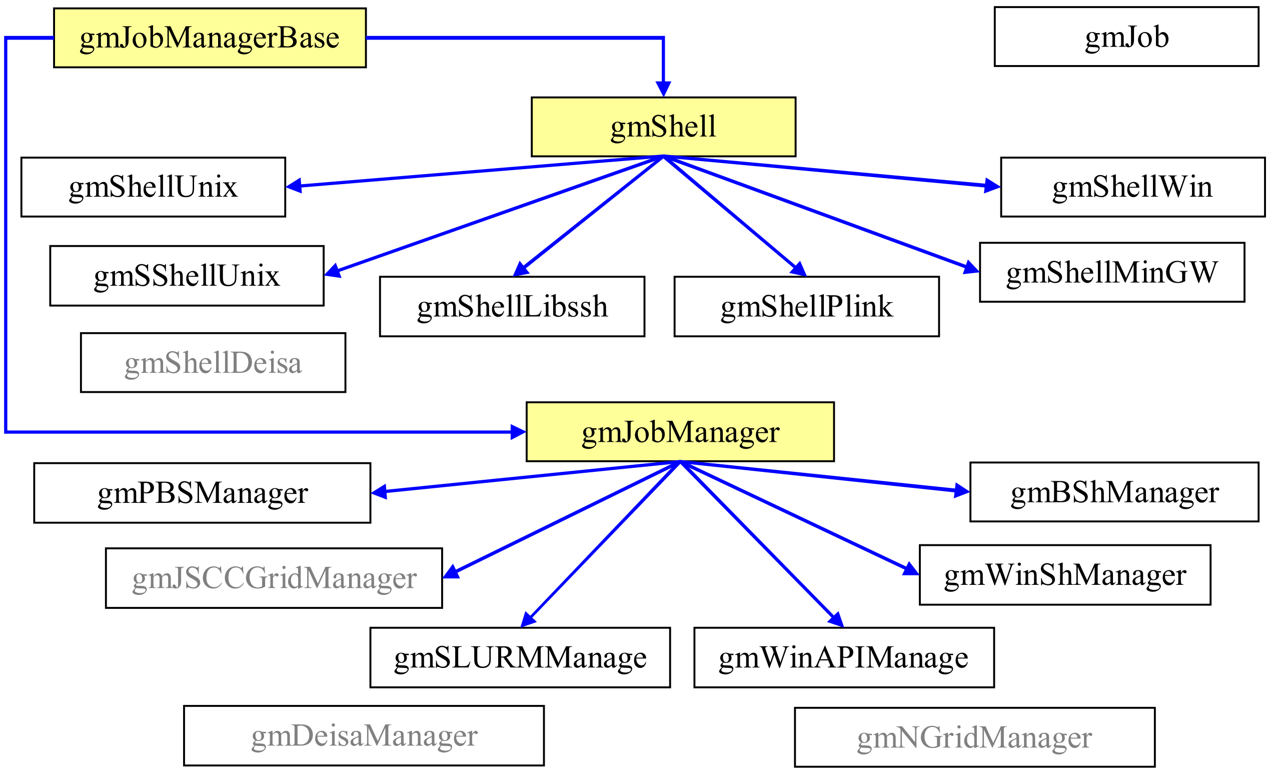
1.1 Terms

- **Resource Manager** is a software installed on the cluster/grid front end which gives an access to the corresponding computational resources: allows one to schedule, execute and control user applications. Resource manager examples are PBS, Globus, Condor, etc.
- **Remote System** is the computer where the resource manager is installed.
- **Job** is a minimal object which can be passed to the resource manager for execution. Usually the job contains a program, input data files and execution parameters.
- **Job Manager** is the GridMD library component represented by a class derived from gmJobManager. This class provides an interface between the GridMD application and a specific resource manager.

1.2 Data types defined in the Job Manager GridMD Component

```
typedef char* pSTR;
typedef const char* pCSTR;
typedef list<gmJob*> JobList;
```

1.3 Job Manager class hierarchy



[*] Abstract classes are shown with the yellow background.

2 Job Manager usage

2.1 Selection of the remote access protocol

In order to initialize a particular Job Manager one should specify the communication protocol which will be used to access the computer where the resource manager is installed. Though it can be the same computer where the GridMD program is executed, we will recall it as a *remote system*. Each protocol is represented by a class derived from gmShell. These classes provide general functionality to copy files to and from the remote system, manage directories, execute remote commands and can be used independently of the resource manager. However the main purpose of the communication protocol class is to access to the resource manager. Below are the prototypes of the class constructors with the arguments related to the specific protocols.

```
    gmShellUnix ()  // defined in defined in jobmngr/unixshell.h
```
Choosing this class implies that the GridMD program runs on the same Unix/Linux machine where the resource manager is installed. The commands are passed to the resource manager through the Unix shell and the files are copied using Unix command line tools.

```
    gmShellUnixSSH (pCSTR userhost, pCSTR ssh_args)  // jobmngr/unixsshell.h
```
The GridMD program runs on a Unix/Linux machine and access the resource manager via SSH protocol; *userhost* contains the remote host name and user name (if needed) in the format "<username>@<hostname>"; *ssh_args* is a string with optional ssh/scp parameters, for instance the following parameter specifies the private RSA/DSA key: "-i <ssh_key_path>". The user should assure that the SSH authentication does not require keyboard-interactive login.

```
    gmShellPlink (pCSTR userhost, pCSTR plink_args)  // jobmngr/plinkshell.h
```
The GridMD program runs on a MS Windows machine and access the resource manager via SSH protocol using plink.exe and pscp.exe tools from the free PuTTY package (http://www.chiark.greenend.org.uk/~sgtatham/putty/); *userhost* contains the remote host name and the user name (if needed) in the format "<username>@<hostname>"; *plink_args* is a string with additional options for plink.exe and pscp.exe, e.g.

```
    -pw <password>           - login with specified password (unsafe)
    -load "<session_name>"   - load settings from saved PuTTY session
    -i <ssh_key_path>        - private key file for authentication
    -C                       - enable compression
```

```
    gmShellDeisa ()  // jobmngr/deisashell.h
```
The GridMD program runs on a MS Windows machine and access the DEISA Grid resource manager via the command line tool called Deisa Shell (DESHL, http://www.deisa.eu/usersupport/user-documentation/deshl). The DESHL package should be pre-installed.

Yet there are other members of gmShell class which are used in some descendant classes:

```
    unsigned gmShell::com_att_num, gmShell::com_retry_delay
```
The variable *com_att_num* defines the number of attempts to establish connection with the remote system, *com_retry_delay* defines a delay between these attempts in milliseconds (used when the first attempt failed). Defaults: *com_att_num* = 1, *com_retry_delay* = 0.

2.2 Working with files and directories

All communication protocol classes described in the previous section have the following functions to manage files and directories.

```
    int gmShell::MkDir(pCSTR remdir)
```
Creates a directory on the remote resource. For Unix/Linux remote systems all parent directories are created as well as "mkdir -p" command is used. The functions returns 0 on success.

```
int gmShell::StageIn( pCSTR locfile, pCSTR remfile, int type = fCopy)
int gmShell::StageIn( const wxArrayString& locfiles, pCSTR remdir,
                      int type = fCopy)
```
Copies a local file or a directory *locfile* to the remote resource using the output name (or directory) *remfile*. The directories are copied recursively. If *type* = gmShell::fMove the source local file/directory is removed. The second function form copies in turn all files/directories from the *locfiles* array (for the details of using wxArrayString class see the wxWidgets library manual at http://docs.wxwidgets.org/stable/wx_wxarraystring.html#wxarraystring). When the *locfile* string starts with the "text:" prefix the line ends are converted from DOS to Unix format. The "text:" prefix can be used only for files (not for directories). If the *remdir* path does not start with '/' it is assumed that the path is related to the user home directory. The functions returns 0 on success.

```
int gmShell::StageOut( pCSTR locfile, pCSTR remfile, int type = fCopy)
int gmShell::StageOut( pCSTR locdir, const wxArrayString& remfiles,
                       int type = fCopy);
```
Copies a the file or a directory *remfile* from the remote system to the local directory using the output name (or directory) *locfile*. The directories are copied recursively. If *type* = gmShell::fMove the source remote file/directory is removed. The second function form copies in turn all files/directories from the *remfiles* array. When the *remfile* string starts with a "text:" prefix the line ends are converted from Unix to DOS format. The "text:" prefix can be used only for files (not for directories). Besides when using the "text:" prefix the *locfile* argument should contain the full local output file path rather than a directory. If the *remdir* path does not start with '/' it is assumed that the path relates to the user home directory. The functions returns 0 on success.

```
int gmShell::Remove(pCSTR remfile)
int gmShell::Remove(const wxArrayString& remfiles)
```
Removing the file or directory (recursively) from the remote system. The second function form removes in turn all files/directories from the *remfiles* array. The functions returns 0 on success.

```
int gmShell::Execute(const wxString& cmd)
int gmShell::Execute(const wxString& cmd, wxString& out, wxString& err)
```
Executing the command *cmd* on the remote system. The second function form captures the outputs to stdout and stderr streams and saves them into the strings *out* and *err*. The functions return the exit code of executed command.

2.3 Job Manager initialization

The job manager initialization starts with the creation of an object of one of the classes derived from gmJobManager listed below. Each class represents the corresponding resource manager. All class constructors take the first argument which references to an existing communication protocol class object. The second argument has different meaning for different classes.

```
gmPBSManager(gmShell& shell, pCSTR service=NULL)    // defined in
                                                    // jobmngr/pbsmngr.h
```
Using Portable Batch System (PBS), *service* is the optional argument which defines additional options for the qsub command, for instance "–q queue@somehost –v MYVAR=MYVAL".

```
gmJSCCGridManager(gmShell& shell, pCSTR service=NULL) // jobmngr/jsccmngr.h
```
Using the Grid system of Joint Supercomputer Center of Russian Academy of Sciences (JSCC), *service* is the Grid gateway host name (x345, x346, x347).

```
gmDeisaManager(gmShell& shell, pCSTR service=NULL) // jobmngr/deisamngr.h
```
Using DEISA Grid manager, *service* is the DEISA site to be used for job submission. When using DESHL all sites should be defined in the config.csv file.

All job managers have the default constructor which can be used along with the Open function having the same *shell* and *service* arguments:

```
gmPBSManager mngr();
mngr.Open(shell, service);
```

Some job managers allows one to set additional configuration parameters using class-specific functions given below.

```
void gmPBSManager::Configure( pCSTR pbs_dir, pCSTR qsub_args,
                              pCSTR job_name_prefix )
```
This function sets the following arguments: *pbs_dir* is the path to the PBS utilities (qsub, qstat, qdel) on the remote system (when non-standard); *qsub_args* has additional qsub options; *job_name_prefix* is a prefix for the PBS job name defined by the "–N" qsub option. If one of the arguments have the NULL value the corresponding parameter remains unchanged.

The base class gmJobManager contains also the following public members.

```
int gmJobManager::wait_timeout;
```
Defines the waiting time in milliseconds used in the functions Wait and FetchResult (see below). If *wait_timeout* = 0 (by default) the waiting period is unlimited.

2.4 Creation of job

In order to define a new job or manage an existing one the user should create the gmJob class object. There are two types of these objects. The so called 'unmanaged' object is created explicitly by the user (in heap or stack) using the constructor:

```
gmJob()
```
Then the user is responsible for freeing the memory allocated by this object.

The second type is the 'managed' object which is created implicitly by the job manager when the user calls

```
gmJob* gmJobManager::CreateJob()
```
The corresponding object is located in the heap and it will be automatically deleted when Clear or Detach functions are called (see below) or when the job manager object is destroyed. The managed objects are initially linked with the manager that created them and cannot be submitted to another manager.

2.5 Preparing job for submission

Job parameters are defined using the gmJob class members given below.

```
void gmJob::AddInFile(pCSTR src, pCSTR workpath = "", unsigned flags = 0)
```
This function defines the name of the input file or directory *src* which will be copied to the working directory on the remote system on job submission. Each job creates the unique working directory on submission to store input, output and temporary files. The source file *src* can be located either on local or on the remote system. If the path to the remote source file does not start with '/' it is assumed to be related to the user home directory. The argument *workpath* may specify the name of output file and/or the path related to the working directory. If *workpath* is empty the source file/directory is cop-

ied to the root of the working directory with the same mane. The directories are copied recursively. The argument *flags* defies additional copying parameters:

`gmJob::REMOTE`	file *src* is located on the remote system (to which the job will be submitted);
`gmJob::TEXT`	the line endings in the given text file will be converted according to the local and remote OS types. This is not applicable to directories;
`gmJob::CREATEPATH`	the new directory with the name given by *workpath* value will be created under the working directory and the source file will be copied to this new directory.

Different parameters can be combined in the *flags* argument by using "|" operation, for example:

```
job.AddInFile("localdir/myfile", "outdir", gmJob::CREATEPATH | gmJob::TEXT);
```

To specify multiple input files the user should call AddInFile function for each of them. The files will be copied in the same order in which the corresponding AddInFile functions are called.

```
void gmJob::AddOutFile(pCSTR dst, pCSTR workpath, unsigned flags = 0)
```

This function defines the name of the output file or directory which path *workpath* is related to the working directory. This file/directory will be copied to the *dst* path on the local or remote system on FetchResult call (see below). The copying parameters are defiled in the same way as for AddInFile. Setting of the flag gmJob::CREATEPATH leads to creation of the output directory with the name given by *dst* on the local or remote system. If the values of *workpath* is "STDOUT" or "STDERR" then the *dst* file will receive the stdout/stderr streams. In this case the *dst* should be name of the file, not directory, and the flag gmJob::TEXT is set automatically. Alternatively the user can manually redirect the stdout/stderr output to a custom file (see *command* variable below).

```
void gmJob::ClearInFileList()
void gmJob::ClearOutFileList()
```

Clear the list of input or output files defined by AddInFile or AddOutFile.

```
wxString gmJob::command;
```

This string contains the Unix shell command which will be executed when the job starts on a working node of the remote system. The command is executed in the working directory created for this particular job. Any input file specified in the argument of AddInFile function will be copied to the same directory and therefore can be executed by the command "./file_name". If multiple commands must be executed they should be delimited by the symbols ';' or '\n'. When a script file is copied from a Windows workstation one usually needs to set the "executable" attribute. In this case the *command* may have the form "chmod u+x script_file; ./script_file". Alternatively one can provide a path to the remote script file which already has the "executable" attribute, e.g. "/usr/bin/perl /my_dir/my_script.pl". Such file need not to be uploaded as an input file of course. The *command* string should not contain single or double quotes. If the quotes are indispensable it is recommended to store the command into a text file, upload it as an input file and use this script in the *command* string. For a MPI-based program the command should include mpiexec call.

```
unsigned gmJob::nproc, gmJob::nthreads, gmJob::walltime;
```

Optional resource manager parameters: *nproc* is the number of requested processes (for parallel programs), *nthreads* is the number of threads per process, *walltime* is the maximum execution time in seconds. By default these class members have zero values which means that the resource manager defaults will be used.

2.6 Job execution and control

In this section the gmJob member functions for job execution and control are considered. During execution the job passes through a sequence of states given in the description of the GetState function.

```
int gmJob::GetState()
```

Returns the current job state. Possible return values are as follows:

```
JOB_INIT          initial state after creation of the gmJob object;
JOB_PREPARED      job is initialized, linked with the Job manager, its work-
                  ing directory on the remote system is created, input files
                  are copied, job description or a starting script for the
                  resource manager is created but not submitted;
JOB_SUBMITTED     job is submitted to the remote resource manager;
JOB_QUEUED        job is in the queue of the resource manager but it is
                  still not running;
JOB_RUNNING       job is running, the intermediate results are available
                  since that;
JOB_SUSPENDED     job is suspended;
JOB_EXITING       job is completed but not all output files are available;
JOB_COMPLETED     job is completed and all output files including stdout and
                  stderr streams are available;
JOB_HAVERESULT    job is completed and all required output files are re-
                  trieved from the working directory;
JOB_FAILED        job submission has failed, job is stopped or its state is
                  unknown.
```

It should be noted that when the job is queued or running on the remote system, call to GetState can cause a delay due to an additional transaction between local and remote systems.

```
int gmJob::LastState()
```

Returns the job state on the end of last operation. This function is similar to GetState but it never causes a delay because no transaction between local and remote systems is performed.

```
static pCSTR StateName(int state)
```

Returns a pointer to the text string describing the state given in the *state* argument.

```
int gmJob::Submit(gmJobManager& mngr, pCSTR user_id=NULL, bool mktemp=true)
int gmJob::Submit(pCSTR user_id=NULL, bool mktemp=true)
```

This function submits the job for execution through the existing job manager *mngr*. For the managed job objects the second version of the function (without *mngr* argument) should be used as the job is liked to the manager on creation. If *mktemp* = true the unique job id is created automatically by using Unix mktemp utility. Additionally the user can specify the custom initial part (prefix) of the id providing the *user_id* argument. If *mktemp* = true the whole id is defined by *user_id* argument and the user has the responsibility to assure that all jobs have different ids. Before job submission a working directory is created on the remote system with the name based on the job id. Then the input files and directories are copied to that directory. It is guaranteed that after return from the Submit all copy operations are completed. The function returns just after submission and does not wait for the job completion. The subsequent job states can be retrieved via GetState function. During Submit execution the *mngr* reference is saved in the gmJob object so that the job becomes linked with the given job manager. If the submission fails or one of the input files can not be copied the job is not executed and its state becomes JOB_FAILED. If the maximum number of submitted jobs is reached the job is switched to the JOB_PREPARED state. Later it may be executed automatically when the manager determines that one of running jobs finishes. If the submission is successful the job state becomes JOB_SUBMITTED. Submit function returns the final job state.

`wxString gmJob::`**`GetID`**`()`

Returns the full job id if this job was submitted, otherwise returns an empty string.

`wxString gmJob::`**`GetWorkDir`**`()`

Returns the working directory path on the remote system for the given job. If the job was not submitted it returns an empty string.

`gmJob* gmJobManager::`**`operator[]`**`(pCSTR id)`

Overloaded operator [] of the job manager allows one to find the job using its id. Usage example: `gmJob *myjob = manager["myjob"]`. If the job is not found the function returns NULL.

`int gmJob::`**`Wait`**`()`

This function waits until the submitted job gets JOB_COMPLETED or JOB_FAILED state or until the maximal waiting time defined by the class member gmJobManager::wait_timeout (in milliseconds) is exceeded. This functions does not copy any output files. Copying of the results can be done by the FetchResult function. The return value is the final job state (similar to GetState()). The exception is thrown if the job is not submitted before the Wait call.

`int gmJob::`**`FetchResult`**`(bool fWait=true)`

Copies the output files specified by the AddOutFile function calls from the job working directory to the local and/or remote system (for the files with gmJob::REMOTE flag). If the job is not completed when FetchResult is called, depending on the *fWait* flag the function either wait for job completion during the period of gmJobManager::wait_timeout like Wait (*fWait*=true) or tries to copy the intermediate output files regerdless of the job state (not supported by some Grid job managers). If the job is completed the FetchResult function changes its state to JOB_HAVERESULT on sucess and to JOB_FAILED on error. If the function is called for an incomplete job its state is changed and copying errors are ignored. FetchResult can be used for the jobs that already have JOB_FAILED state. The return value is the final job state, the exception is thrown if the job is not submitted.

`int gmJob::`**`StageOut`**`(pCSTR dst, pCSTR workpath, unsigned flags = 0)`

Copies a single file/directory *workpath* from the job working directory to the local or remote (for the files with gmJob::REMOTE flag) path *dst*. Usually this function is used to obtain an intermediate result during job execution. However not all job managers allows one to access the working directories until the job is completed. The function returns 0 on success, ≠ 0 on copying error and throws an exception if the job is not submitted. If the value of *workpath* is “STDOUT” or “STDERR” the *dst* file will receive the program output to the stdout/stderr stream. In this case the *dst* should be name of the file, not directory, and the flag gmJob::TEXT is set automatically.

`void gmJob::`**`Stop`**`()`

Stops the job and changes its state to JOB_FAILED. The function throws an exception if the job is not submitted.

`void gmJob::`**`Detach`**`()`

Detaches (unlinks) the gmJob object from the job manager but does not stop the real job running on the remote resource. The managed job object is destroyed on Detach whereas the unmanaged one is just switched to JOB_INIT state. Later the detached unmanaged object can be destroyed by the user or used to start another job which will have no effect on the original detached job. On deletion of the gmJob object the Detach function is invoked automatically.

`void gmJob::Clear()`

Stops the job if it is running, removes its working directory on the remote system and unlinks it from the job manager. The managed job object is destroyed whereas the unmanaged one is switched to JOB_INIT state and, if needed, can be used to start another job using another manager. The function clears the internal job fields but the user defined class members (*command, nproc, nthreads, walltime*, etc) and input/output file lists.

`gmJob* gmJobManager::Restore(pCSTR id)`

The job manager searches the job with the given id which have been submitted earlier using the same job manager and on the same remote system. If the job is found a managed gmJob object is created and its address is returned, otherwise the function returns NULL. Usually this functions is used in conjunction with Detach to restore the jobs using their ids. The job can be restored even if it was run by another process.

`int gmJob::Attach(gmJobManager& mngr, pCSTR id)`

The function restores the job with the given id in the same way as Restore but the information about the job is stored into the given unmanaged gmJob object. The return value is the latest job status or JOB_INIT if the job is not found. If the job is linked with any job manager (was submitted) before calling Attach an exception is thrown.

2.7 Batch job processing functions

The functions described in this section are the members of a jobs manager class and they are intended to perform an action on a set of jobs linked to this job manager. All these functions have the argument *id_prefix* which nonzero value indicates that only the jobs which id start with the given prefix will be processed. If *id_prefix* = NULL all jobs will be processed. In most cases the execution of a batch processing function is analogous to the successive run of the corresponding serial functions described in the previous section.

`int gmJobManager::WaitAll(pCSTR id_prefix = NULL)`

Waits for completion of the given jobs and returns one of the following: JOB_COMPLETED if all jobs are completed successfully, JOB_FAILED if one of the jobs fails, JOB_INIT is no job with the given id prefix is found.

`int gmJobManager::FetchAll(bool fWait=true, pCSTR id_prefix = NULL)`

Copies output files to the local and/or remote system for the given jobs and returns: JOB_HAVERESULT if the results of all jobs are copied successfully, JOB_FAILED is one of the jobs or corresponding copy operation fails.

`void gmJobManager::StopAll(pCSTR id_prefix = NULL)`

Stops all given jobs.

`void gmJobManager::ClearAll(pCSTR id_prefix = NULL)`

Stops all given jobs, remove their working directories, changes their state to JOB_INIT and unlinks them from the job manager.

`void gmJobManager::DetachAll(pCSTR id_prefix = NULL)`

Unlinks the given jobs from the job manager. All managed job objects are destroyed. This function is invoked automatically on the gmJobManager deletion.

`JobList gmJobManager::RestoreAll(pCSTR id_prefix = NULL)`

Searches the jobs with the given *id_prefix* which were submitted earlier using the same job manager class and on the same remote system (see Restore). For each found job a new managed gmJob object is created and linked with the manager. The function does not create duplicates for the jobs which are already linked with this manager. The return value is the STL list class object (see the definition of the JobList data type) which contains pointers to all newly created gmJob objects. These objects are deleted (deallocated) automatically when Detach or Clear is called or when the job manager is deleted.

```
    JobList gmJobManager::GetJobList(pCSTR id_prefix = NULL)
```

This function returns the STL list class object whose elements are the pointers to all the jobs with the given *id_prefix* linked with this jobs manager. Both managed and unmanaged jobs are returned. As the entire list is copied, changing it does not affect the job manager internal job list.