The Task Manager for the LHCb On-Line Farm

LHCb Technical Note

Issue:	1
Revision:	1
Reference:	LHCb 2004-099 DAQ
Created:	3 Nov. 2004
Last modified:	18 Aug. 2005

Prepared By: F. Bonifazi, D. Bortolotti, A. Carbone, D. Galli, D. Gregori, U. Marconi, G. Peco, V. Vagnoni.

Abstract

The Task Manager is a utility to start, stop and list processes on the on-line farm. Each process started by the Task Manager has a string environment variable set, named UTGID (User defined unique Thread Group Identifier) which allows identifying the process. The Task Manager uses the UTGID to list the running processes and to identify the processes to be stopped. It has also the ability to start a process using a particular user name and to set the scheduler type and the priority for the process itself. The Task Manager package includes a Linux DIM server (`tmSrv`), four Linux command line DIM clients (`tmStart`, `tmLs`, `tmKill` and `tmStop`) and a PVSS DIM client.

Document Status Sheet

Table 1 Document Status Sheet

1. Document Title: The Task Manager for the LHCb On-Line Farm			
2. Document Reference Number: LHCb 2004-099 DAQ			
3. Issue	**4. Revision**	**5. Date**	**6. Reason for change**
2	2	18 Aug. 2005	Version related with FMC-1.6 software
1	1	22 Nov. 2004	First released version
0	Draft	11 Nov. 2004	First version

Table of Contents

List of Figures

1. Requirements

During the operation of the on-line farm, several processes have to be started and kept running on the computing nodes and on the sub-farm controllers, including the event builder, the trigger-related processes and the monitoring processes. One can foresee that these processes could also need to be restarted, not only in case of lock or runtime errors, but also if a change in the trigger algorithm configuration is requested.

It is therefore needed a task management system which allows to monitor the processes running on the farm nodes and to stop and/or restart them, following a central command or an automatic control procedure.

We explicitly distinguish between the *Task Manager System*, which has the mere ability to start and stop a process and to list the processes started on any node of the farm on the one hand, and on the other hand the *Process Controller System*, which has the responsibility to keep a process running, restarting it (by means of the Task Manager) in case of abnormal process termination. This note concerns only the Task Manager System, while the Process Controller System will be a subject of a separate note.

The Task Manager System must be able to *keep track of the started processes* and distinguish each other, even if they are different instances of the same executable image, in order to be able to monitor and, if needed, to stop them properly.

The Task Manager must also be able to set the *scheduling policy*, the *priority* and the *process-to-CPU affinity* of the started processes. As a matter of fact, the on-line farm must manage a very large amount of data and therefore must operate in real-time mode. For example, the L1 process must respond with a very low latency and therefore must never be pre-empted by the HLT process. This can be achieved by running both the processes with the real-time FIFO scheduler and assigning to the L1 process a higher priority than the HLT one. Besides, to optimize L1 processes performance, it is desirable that the process remains stuck to the same CPU in multiprocessor nodes, in order to avoid, as far as possible, context switches and register pop out.

Moreover, the Task Manager must be able to set the UID (user identifier) of the started process to a user different from root. In fact, the started processes don’t need to have root privileges, which could be dangerous for unwanted actions which could damage the system installation. Root privileges are usually necessary to run a process at a high priority or to use a real-time scheduler; however this target can also be achieved by starting a process as root and changing then the UID, after the scheduler/priority has been set.

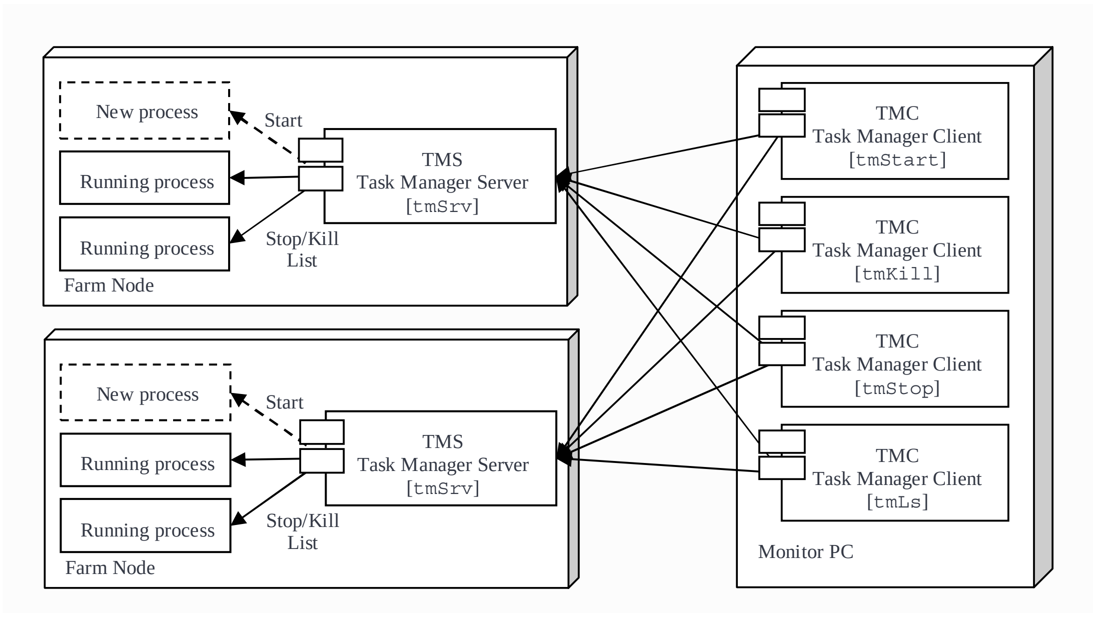


Figure 1. The Task Manager deployment.

Furthermore the Task Manager must be able to *redirect the standard error and/or the standard output* of the started process to a logger in order to retrieve error messages sent by the started application. Even if it is always better that the application itself sends its messages to a logging facility instead of `stderr/stdout`, nevertheless we must remember that important debugging messages are sent to `stderr` by system utilities like the dynamic linker (e.g.: "`/opt/SFM/tm/examples/counter: error while loading shared libraries: libdim.so: cannot open shared object file: No such file or directory`").

Finally, the Task Manager must react to a process termination, by updating immediately the published process list (to allow the Process Controller to restart the process immediately, if requested) and logging a message with the *exit status* or the *number of signal* that caused the child process to terminate.

In this paper, following the Linux threads implementation, we use the term "*task*" to design a general scheduling entity or a general execution flow (an instance of a program in execution), which could be a stand-alone process or a lightweight process (i.e. a thread of a multithreaded process). The terms "process" and "thread group" are interchangeable in this paper as are the acronyms PID (Process Identifier) and TGID (Thread Group Identifier), due to the fact that, since kernel 2.4, processes are implemented as thread groups.

2. Implementation

2.1. DIM

The Task Manager is based on the DIM (Distributed Information Management System) inter-process communication layer [1]. DIM has client/server architecture and uses a Name Server mechanism to publish/subscribe services.

Each farm node runs a Task Manager Server (TMS, whose executable name is `tmSrv`), which registers commands (process start, kill and stop) and services (process list) with the DIM Name Server and makes them available to the control client.

The client asks the DIM Name Server which server makes the required commands and services available, then contacts directly the server to subscribe to services (to bring itself up-to-date about the process list) and requires command execution.

2.2. The UTGID mechanism

In order to be able to monitor and to stop or restart the running processes, the task management system must be able to keep track of the started processes.

The Linux operating system identifies processes (for scheduling purposes) using the integer number TGID (Thread Group Identifier, the thread-epoch version of the old PID, Process Identifier). Associated with the TGID there is the COMMAND string, that is the file name of the process executable image.

None of these two identifiers, taken separately, is however suitable for process tracking: the TGID is an integer number assigned sequentially, so that the same process is assigned a different TGID when it is restarted; COMMAND on the other hand is the same for two different instances of the same process. For process tracking purposes a process could be identified by a combination of TGID and COMMAND, but such a combination does not help to remember which work the different instances of COMMAND are doing.

An alternative approach consists in assigning one more identifier to each process started by the TMS, called UTGID (User assigned unique Thread Group Identifier). UTGID is a string (possibly but not necessarily composed by the COMMAND string and an instance counter) which must be unique on a PC: the same UTGID cannot be used for more than one process on the same PC.

In multithreaded processes, the same UTGID environment variable is set for all the threads of the process; however only thread groups (i.e. processes) are listed by the list service and only thread groups are sent signals.

2.3. The process environment

The UTGID string should be stuck to its process (in order to survive the TMS) but must be accessible by the TMS from outside the process (once the process has been started), because the TMS must be able to associate a TGID to the UTGID to stop a process (the `kill(2)` system call takes the TGID as argument).

The UTGID string is therefore stored by the Task Manager among the process environment variables.

When a process is started, in the *initial process stack*, together with the command line arguments array (`char **argv`), each process is also passed an *environment list*, which is a null-terminated array of character pointers to null-terminated C strings. All these environment null-terminated C strings are stored in contiguous memory locations, terminated by a null string (see Figure 2). The address of the array of pointers is stored in the global variable `char **environ`, while the location of the first character of the contiguous strings is referenced by the i-node `/proc/<tgid>/environ` (see Figure 2). By convention the environment consists of strings having the format: “`NAME=VALUE`”.

The process environment variables can be accessed in 4 different ways in the Linux OS:

1. Through the call to the library functions: `putenv(3)`, `setenv(3)`,

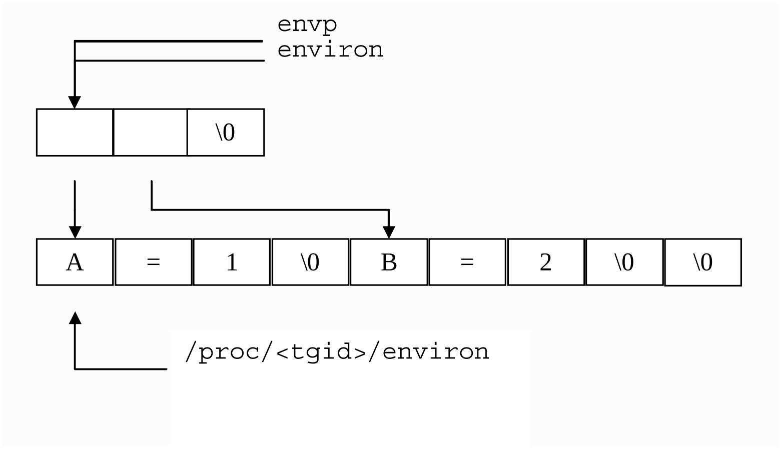


Figure 2. The Process environment.

`clearenv(3)`, `unsetenv(3)`, `getenv(3)` from inside the process itself.

2. Through the global variable: "`extern char **environ`" (see manual pages for `environ(5)`) from inside the process itself;

3. Through the third argument (`envp`) of the main function: `int main(int argc, char **argv, char **envp)` from inside the process itself;

4. From the i-node `/proc/<tgid>/environ` from inside and outside the process by all the processes which are running on the computer having the same UID of the process or the UID equals to 0 (`root`).

Although anyone of these four methods can be used, in principle, to access the process environment variables, a certain caution must however be used. As a matter of fact, the environment seen by methods (1), (2) and (3) can be modified by calling `putenv(3)` or `setenv(3)`, or, equivalently, by allocating in the stack a new string (using `alloca(3)`) and making `envp/environ` pointing to the new string, while the environment seen using method (4) comes out unmodified. Only upon the execution of `execve(2)` system call the initial process stack is rewritten, so that the environment strings pointed by `envp/environ` became again contiguous in memory and the environment seen by the methods (4) became the same seen by the other methods.

This makes **impossible the change of the environment variables referenced by `/proc/<tgid>/environ` from inside a running program**. The only way to set an environment variable stored in `/proc/<tgid>/environ` in a new process is to call `setenv(2)` between the `fork(2)` and `execve(2)` system calls at process start-up.

2.4. Starting a process

The creation of a new task in Linux is achieved by the `fork(2)`, `vfork(2)` and `clone(2)` system calls.

The `clone(2)` system call is used to create processes which need to share parts of their execution context with the calling process, such as the memory space, the table of file descriptors, and the table of signal handlers; it is therefore suited to create lightweight processes.

The `vfork(2)` system call is used to create new processes without copying the page tables of the parent process and is therefore suited to create processes which immediately issue an `execve(2)` in performance sensitive interactive applications (since process start is faster). The limitation of `vfork(2)` is the fact that the parent execution is suspended until the child makes a call to `execve(2)` or `_exit(2)`. The `vfork(2)` call is also tricky to use (details of the signal handling are obscure) and its usage is discouraged; moreover its advantage in process starting performance is not so sensible under Linux, since `fork(2)` is implemented using copy-on-write pages (virtual memory will only be copied when one of the two processes tries to write to it; any virtual memory that is not written to, even if it can be, will be shared between the two processes without any harm occurring), so that the only penalty incurred by `fork(2)` is the time and memory required to duplicate the parent's page tables and to create a unique task structure for the child (under Linux, `fork(2)` does not require to make a complete copy of the caller's data space).

The `fork(2)` system call is therefore the most appropriate to create a new heavyweight process and to set its process environment (broadly speaking) before issuing the `execve(2)` call.

Several settings have to be made between the `fork(2)` and the `execv(3)` call (the library call which in turn invoke the `execve(2)` system call) by the TMS as can be seen in Figure 3:

1. If required, the environment of the new process must be cleared in order that it does not inherit from the parent useless environment variables [`clearenv(3)` library call].

2. Some process-specific environment variables must be set [`putenv(3)` library call].

3. The UTGID environment variable must be set [`setenv(3)` library call].

4. All open file descriptors must be closed [`close(2)` system call]. The standard file descriptor `STDIN_FILENO` must then be re-opened on `/dev/null`. The standard file descriptors `STDOUT_FILENO` and `STDERR_FILENO` must either be re-opened on `/dev/null` (to discard every write performed by the process) or, if required, be re-opened on the DIM logger's FIFO (to redirect output/error messages to the DIM logger utility) [`open(2)` and `dup(2)` system calls].

5. The process-to-CPU affinity mask must be set [`sched_setaffinity(2)` system call].

6. The scheduling policy required to run the process must be set. If the chosen scheduler is a real-time scheduler, the static priority of the process must be set [`sched_setscheduler(2)` system call].

7. If the process is scheduled by the default time-sharing scheduler, the nice level (used to compute the dynamic priority of the process) must be set [`setpriority(2)` system call].

8. The user identifier of the user which will be the owner of the process must be set [`setuid(2)` system call].

If the process has to be started as a daemon process, additional settings have to be made:

1. The umask of the process must be reset [`umask(2)` system call]. The umask modifies the file permissions of the newly-created files, which is set to the bitwise AND of the permission set with the `open(2)` call with the bitwise negation of the umask.

2. Create a new process session (process group) and set the process as a process group leader [`setsid(2)` system call].

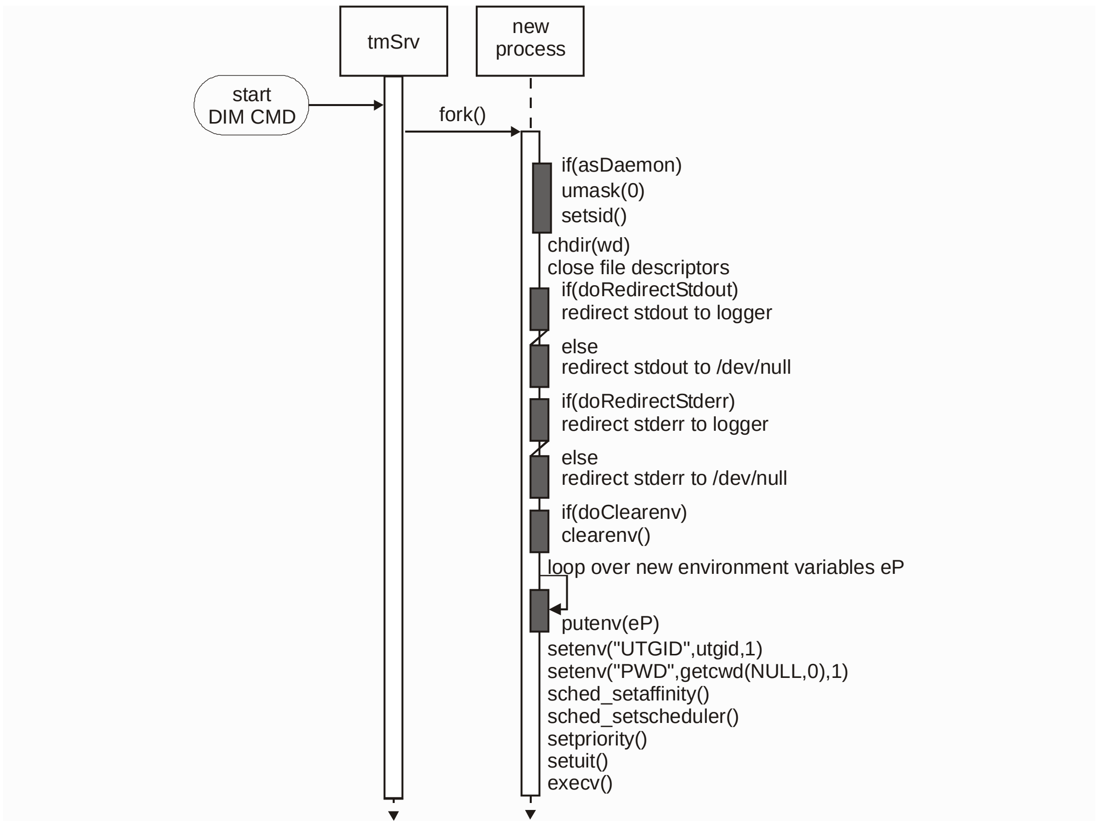


Figure 3. Sequence diagram of the start procedure.

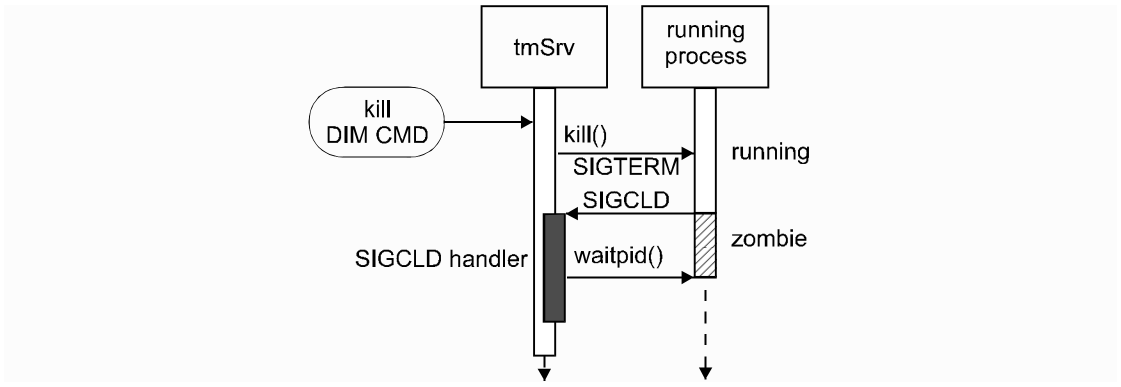


Figure 4. Sequence diagram of the kill procedure. The running process does not catch the SIGTERM signal.

2.5. Sending a signal to a process

Usually a signal is sent to a process either to stop it or to trigger a reload of one or more configuration files. In Linux, signals to other processes are sent using the `kill(2)` system call. If the process implements a signal handler, it is executed asynchronously on signal reception (the handler could trigger a configuration reload or gracefully terminate the process), otherwise, if the signal is not masked, the process terminates.

Upon process termination, since the TMS (the parent process) remains alive, the exiting process turns into a "zombie" process and sends a `SIGCHLD` signal to TMS. TMS, in turn, implements a `SIGCHLD` signal handler which calls `waitpid(2)` to release all the resources used by the process.

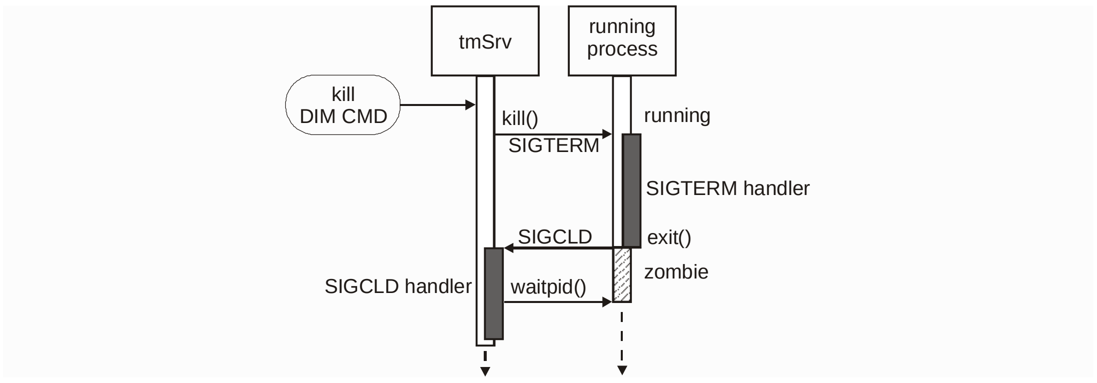


Figure 5. Sequence diagram, of the kill procedure. The running process catches the SIGTERM signal and exit gracefully.

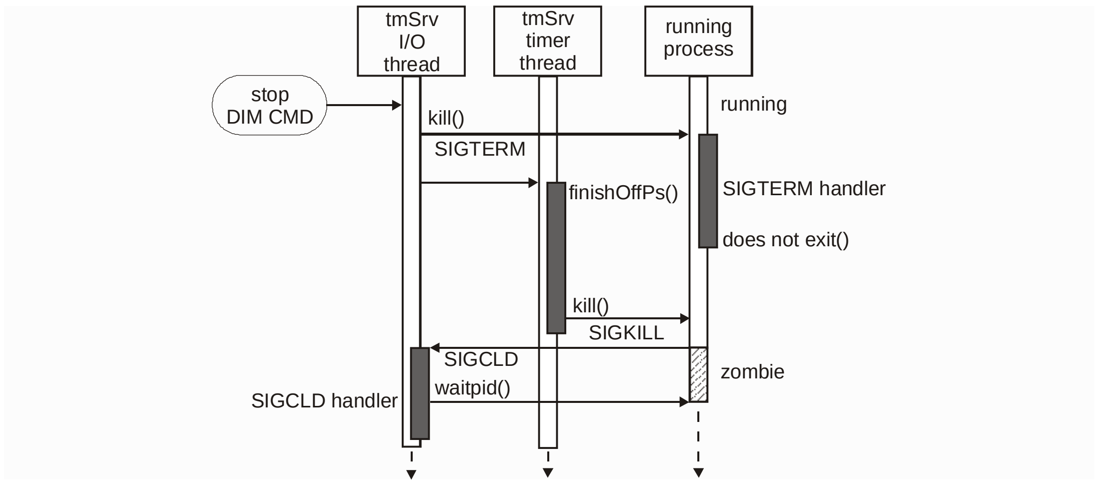


Figure 6. Sequence diagram of the stop procedure. The running process catches the SIGTERM and continue, but is stopped by the delayed SIGKILL.

2.6. Stopping a process

The stop command handler on the TMS is designed to stop a process in the most general way. It sends a signal to the process (default `SIGTERM`, but another signal can also be sent) and schedules the deferred execution of the `finishOffPs()` function.

The `finishOffPs()` function controls whether the process is already dead, otherwise sends a second signal (this time a `SIGKILL` signal) to the process. This way the process is left the chance to exit gracefully on a `SIGTERM` reception, but, if it fails, it is stopped abruptly by a `SIGKILL` signal after a certain delay.

2.7. Listing the running processes

The list service handler in TMS is designed to get an updated list of the running processes which have the UTGID defined. The list is obtained by the TMS, by accessing all the i-nodes `/proc/<tgid>/environ` and looking at them for the UTGID variables.

The list published by TMS is updated not only periodically (every 10 seconds), but also whenever a TMS command is executed and one second after a TMS command has been executed.

The published list is also updated on spontaneous process termination (i.e. a process termination not triggered by the Task Manager).

By means of this service, the Process Controller is able to check if the processes listed in the configuration database are running and, if they are not, to start or restart them.

2.8. Logging and list updating on process termination

When a process (started by the TMS) terminates, either spontaneously (`main()` function's return, `exit(3)` call, exception, signal reception, etc.) or because it is killed by the Task Manager, the TMS process (which is its parent process) remains alive, so that the exiting process is turned into a "zombie" process and a `SIGCHLD` signal is sent to TMS.

The TMS, in turn, implements a `SIGCHLD` signal handler which:

- calls `waitpid(2)` to free all resources used by the process and to expunge it from the process list;
- sends a message to the logger facility with the *exit status* or the *number of signal* that caused the child process to terminate;
- updates immediately the published process list (to allow the Process Controller to restart the process immediately if requested).

2.9. The Task Manager Server's threads

Like all the DIM servers, the TMS process is multi-threaded and is composed of 3 light-weight processes (threads): the *main thread* (which runs the main control loop that schedules list service updates and cleans-up "zombies"), the *I/O thread* (which manages DIM commands and services) and the *timer thread* (which manages timers and delayed executions).

The scheduling policy for the TMS threads and the static (real-time) priority of each TMS thread can be set through the TMS command line, using the switches: `--schedpol`, `--mainprio`, `--ioprio` and `--timerprio`.

While there are no particular prescriptions in running the TMS with the Linux standard time-sharing scheduler (the static priority of the three threads cannot be different from zero and the started processes cannot be run with real-time schedulers), more attention must be put in running the TMS with real-time schedulers (fifo or round-robin): the I/O thread will probably have the highest priority, the main thread will probably have an intermediate priority and the timer thread will probably have the lowest priority. By default, TMS is run using the real-time round-robin scheduler and the static priorities are set to 80 (main thread), 93 (I/O thread) and 10 (timer thread).

The TMS forbids spawning a process with a static priority equal or higher than the TMS I/O thread, to avoid starting a process that the TMS could not be able to kill (e.g., a process which never relinquish the CPU, running with fifo scheduler and a static priority equal or higher than the TMS, does never give the TMS the chance to kill it). Thus a TMS running using the time-sharing scheduler (zero static priority) cannot start a real-time process (static priority greater than zero) .

2.10. Processes I/O redirection

The processes started by the Task Manager run in *background* on the farm nodes and *cannot perform terminal I/O*. If a process, started by the TMS, try to write a message to STDOUT_FILENO or to STDERR_FILENO (e.g., using a `printf(3)`), by default messages are sent to `/dev/null`, and therefore are lost forever. The `-o` and `-e` switches on the command string processed by the TMS start service allow instead redirecting these messages to the DIM logger.

Even if it is always better that the application itself sends its messages directly to the DIM logger (using the DIM logger API), instead of writing to `stderr/stdout`, nevertheless we must remember that important debugging messages are sent to `stderr` by system utilities like the dynamic linker (a typical error message from the dynamic linker is, e.g., "`/opt/SFM/tm/examples/counter: error while loading shared libraries: libdim.so: cannot open shared object file: No such file or directory`").

3. The Task Manager Server (TMS)

The TMS, whose executable image name is `tmSrv`, runs on each node of the sub-farm. It is *recommended* to *run the Task Manager as user* `root` and to *limit the root access* as desired *by means of* `–p` *flag*. Running the Task Manager as a user different from root allows only starting processes as such a user and listing and stopping processes owned by such a user. The Task Manager could be started by `init` process, using the `respawn inittab` option, in order to insure that it is always alive.

3.1. Synopsis

```
tmSrv [-l logger][-p permission][-d default_user]
      [--schedpol scheduling_policy]
      [--mainprio main_thread_priority]
      [--ioprio I/O_thread_priority]
      [--timerprio timer_thread_priority]

tmSrv [-h]
```

3.2. Description

Starts the TMS on the node and sends its log messages to a logger.

3.3. Command line options

`-h` Print the program usage and exit immediately.

`-l logger`
Use **`logger`** as error logger. Values allowed for **`logger`** are in the range 0...7. The value is the result of a bitwise `OR` of the following values:

0x0	`NOLOG`	Don't write log at all (default).
0x1	`DIMLOGGER`	Send log to DIM logger.
0x2	`STDERRLOG`	Send log to stderr.
0x4	`DIMSVC`	Send log to a specific DIM service.

`-d default_user`

Starts the processes as the user **`default_user`**, by default. If **`permission`** is greater then 0 (i.e. if **`permission`** is equal to 1 or 2), using the **`-n`** DIM command option, a process can also be started as a user different from **`default_user`**.

`-p permission`

Allowed values for **`permission`** are in the range 0...2. The meaning of this parameter is the following:

0. Processes can only be started as the user **`default_user`**.

1. Process can be started as a user different from **`default_user`** (by using **`-n`** DIM command option) but not as the user **`root`**.

2. Process can be started as a user different from **`default_user`** (by using **`-n`** DIM command option) including the user **`root`**.

`--schedpol scheduling_policy`

Run the TMS using the scheduling policy **`scheduling_policy`**. Allowed values for **`scheduling_policy`** are: 0 (`SCHED_OTHER`, the standard Linux time sharing scheduler), 1 (`SCHED_FIFO`, the real-time fifo scheduler) and 2 (`SCHED_RR`, the real-time round-robin scheduler). Default value: 2.

`--mainprio main_thread_priority`

Run the TMS with the priority of the main thread (the thread which runs the main control loop that schedules list service updates and cleans-up "zombies") equal to **`main_thread_priority`**. Allowed values: 0 for `SCHED_OTHER` scheduling policy, 1…99 for `SCHED_FIFO` and `SCHED_RR` scheduling policy. Default value: 80.

`--ioprio I/O_thread_priority`

Run the TMS with the priority of the I/O thread (the thread which manages DIM commands and services) equal to **`I/O_thread_priority`**. Allowed values: 0 for `SCHED_OTHER` scheduling policy, 1…99 for `SCHED_FIFO` and `SCHED_RR` scheduling policy. Default value: 93.

`--timerprio timer_thread_priority`

Run the TMS with the priority of the timer thread (the thread which manages timers and delayed executions) equal to **`I/O_thread_priority`**. Allowed values: 0 for

`SCHED_OTHER` scheduling policy, 1…99 for `SCHED_FIFO` and `SCHED_RR` scheduling policy. Default value: 10.

3.4. Published DIM command and services

3.4.1. CMD: `/<HOSTNAME>/task_manager/start`

1. Command String Synopsis

```
"[-c][-D NAME=value...][-d][-s scheduler][-p nice_level]
[-r rt_priority][-a cpu_num...][-n user_name][-u utgid]
[-w wd][-e][-o] path [arg...]"
```

2. Description

Start a new process on the node, using the executable file located in **`path`** and the arguments specified in **`arg`**. In the environment of the started process `tmSrv` puts a new string variable, called **UTGID** (User assigned unique Thread Group Identifier).

By default, before starting the process, all file descriptors are closed and standard file descriptors (**`STDIN_FILENO`**, **`STDOUT_FILENO`** and **`STDERR_FILENO`**) are reopened on **`/dev/null`**.

UTGID can be defined by the user (using **`-u`** option) or can be generated automatically (omitting **`-u`** options) by appending to the command name (the name of the executable image) an underscore followed by an instance counter.

3. Options

`-c` Clear the process environment. If this flag is specified, the process environment is cleaned and only the variable UTGID (set as specified in option **`-u`**) and PWD (pointing to the working directory specified with option **`-w`**) are set. If this flag is not specified, the environment of `tmSrv` process is inherited, the variable PWD is changed to point to the working directory specified with option **`-w`**, and the variable UTGID (set as specified in option **`-u`**) is added.

`-D NAME=value`

(Repeatable). Set the environment variable `NAME` to the value `value`. More than one of these options can be present in the same command. They have to be used to set the environment variables which are specific for the particular starting process. Environment variables which are common for many processes can be set, more conveniently, in the `tmSrv` environment (see sections 3.5 and 3.6).

`-d` Run process as daemon. If this flag is specified, process umask is changed into 0 and program is run in a new session as process group leader (**`setsid()`**).

`-s scheduler`

Set **`scheduler`** as the scheduler for the process. This option can also be specified for processes which run as a user different from **`root`** (in fact scheduler is set before uid). Allowed values for **`scheduler`** are (see `sched_setscheduler(2)` manual page for more details):

0 (`SCHED_OTHER`)	The default time-sharing Linux scheduler, with a dynamic priority based on the nice level.
1 (`SCHED_FIFO`)	The static-priority real-time fifo scheduler, without time slicing. A SCHED_FIFO process runs until either it is blocked by an I/O request, it is preempted by a higher priority process, or it calls sched_yield().
2 (`SCHED_RR`)	The static-priority real-time round-robin scheduler. It differs from SCHED_FIFO because each process is only allowed to run for a maximum time quantum. If a SCHED_RR process has been running for a time period equal to or longer than the time quantum, it will be put at the end of the list for its priority.

`-p nice_level`

Set the nice level of the process to **`nice_level`**. This value is used by the SCHED_OTHER time-sharing Linux scheduler to compute the dynamic priority. Allowed values for **`nice_level`** are in the range –20...19 (–20 corresponds to the most favorable scheduling; 19 corresponds to the least favorable scheduling). Nice level may be lowered also for processes which run as a user different from **`root`** (in fact the **`nice_level`** is set before uid). See manual pages for `nice(1)` and `setpriority(2)` for details).

`-r rt_priority`

Set the static real-time priority of the process to **`rt_priority`**. Only value 0 for **`rt_priority`** is allowed for scheduler SCHED_OTHER (the default time-sharing Linux scheduler). For SCHED_FIFO and SCHED_RR real-time schedulers, allowed values are in the range 1...99 (1 is the lowest priority, 99 is the highest priority). See manual pages for `sched_setscheduler(2)` for more details.

`-a cpu_num...`

(Repeatable). Add the CPU **`cpu_num`** to the process-to-CPU affinity mask. More than one of these options can be present in the same command to add more than one CPU to the affinity mask. Started process is allowed to run only on the CPUs specified in the affinity mask. Omitting this option, process is allowed to run on any CPU of the node. Allowed **`cpu_num`** depend on the PC architecture: e.g., in a single-processor PC with Huper-Threading activated or in a dual processor PC without Hyper-Threading **`cpu_num`** can be 0 or 1; in a dual processor PC with Hyper-Threading activated **`cpu_num`** can be 0, 1, 2 or 3.

`-n user_name`

Set the effective UID, the real UID and the saved UID of the process to the UID of user **`user_name`**. If `tmSrv` was started with "**`-p 0`**" command line option only **`default_user`** can be specified as **`user_name`**. If `tmSrv` was started with "**`-p 1`**" command line option, any existing user name different from **`root`** can be specified as **`user_name`**. If `tmSrv` was started with "**`-p 2`**" command line option, any existing user name including **`root`** can be specified as **`user_name`**.

`-u utgid`

Set the string **`utgid`** as the process UTGID (User assigned unique Thread Group Identifier). The UTGID is set as a process environment variable, accessible, from inside the process using **`getenv(3)`** library call, global variable **`environ`** (see `environ(5)` manual page) or **`envp`** argument of program's `main()` function, and from outside the process, using the i-node **`/proc/<TGID>/environ`**.

`-w wd`

Set the string **`wd`** as the process working directory. File open by the process without path specification are sought by the process in this directory.

`-e`

Redirect the standard error to the DIM logger. Omitting this option, the standard error is thrown in `/dev/null`.

`-o`

Redirect the standard output to the DIM logger. Omitting this option, the standard output is thrown in `/dev/null`.

3.4.2. CMD: `/<HOSTNAME>/task_manager/kill`

1. Command String Synopsis

```
"[-s sig] utgid_pattern"
```

2. Description

Send the signal **`sig`** (default: signal 15, i.e. SIGTERM, if the **`-s`** flag is omitted) to all the processes whose UTGID matches the POSIX.2 wildcard pattern **`utgid_pattern`** on the node.

3. Options

`-s sig`

Send the signal sig. If not specified signal 15 (SIGTERM) is sent.

3.4.3. CMD: `/<HOSTNAME>/task_manager/stop`

1. Command String Synopsis

```
"[-s sig][-d delay] utgid_pattern"
```

2. Description

Send the signal **`sig`** (default: signal 15, i.e. SIGTERM, if the **`-s`** flag is omitted) to all the processes whose UTGID matches the POSIX.2 wildcard pattern **`utgid_pattern`** on the node. If some of the processes which receive the signal are still alive after **`delay`** seconds, a signal 9 (SIGKILL) is sent to them.

3. Options

`-s sig`

Send the signal sig. If not specified signal 15 (SIGTERM) is sent.

`-d delay`

Use **`delay`** as the delay between the first signal (SIGTERM or signal specified with **`-s`** option) and the second signal (SIGKILL). If not specified a one second delay is assumed.

3.4.4. SVC: `/<HOSTNAME>/task_manager/list`

1. Description

Return an array of NULL-terminated strings, containing the list of the processes which have the UTGID environment variable set. If there are no running processes with the UTGID variable defined, the service returns the string "`(none)`". The list is updated not only periodically (every 10 seconds), but also whenever a TMS command is executed, one second after a TMS command has been executed and immediately after a process started by the current TMS has terminated.

3.4.5. SVC: `/<HOSTNAME>/task_manager/log`

1. Description

This service publishes DEBUG/INFO/WARNING/ERROR/FATAL messages sent by the task manager.

This service is published only if the TMS is started with the option `-l 4`, `-l 5`, `-l 6` or `-l 7`.

3.4.6. SVC: `/<HOSTNAME>/task_manager/server_version`

1. Description

This service publishes the RCS Identification string of the TMS main program source file, containing version and last modification time.

3.4.7. SVC: `/<HOSTNAME>/task_manager/actuator_version`

1. Description

This service publishes the RCS Identification string of the TMS actuator function source file, containing version and last modification time.

3.4.8. SVC: `/<HOSTNAME>/task_manager/success`

1. Description

This dummy service is always returns 1. It is used by PVSS-DIM to check if `tmSrv` process is running.

3.5. Environment

The program `tmSrv` needs the two environment variables:

`DIM_DNS_NODE`

hostname.domain of DIM dns node.

`LD_LIBRARY_PATH`

Variable, in PATH format, which must contain the path of the shared libraries libdim.so, libSFMutils.so and libproc-3.2.3.so.

The Task Manager environment is inherited by all the processes started by the task manager, unless the **-c** option is specified as start argument.

Therefore all common environment variable needed by many processes started by the task manager has to be in `tmSrv` environment. Process-specific environment variable can be set using "**–D**" command option (see section 3.4.1.3).

A further environment variable

deBug

the debug level.

can be set to 1 or more to make the TMS sending to the DIM logger more debugging messages.

3.6. Examples

The task manager server `tmSrv` can be started using the `inittab`, writing in `/etc/inittab` an entry like:

```
<Id>:<run_level>:respawn:/opt/SFM/sbin/startTmSrv.sh
```

where `<Id>` is a unique sequence of 1-4 characters which identifies an entry in inittab, `<run_level>` lists the run-levels for which the Task Manager have to run, and `startTmSrv.sh` is a shell script like this:

```
#!/bin/sh
DIM_DNS_NODE=lhcbos1.lhcb-bo.infn.it
LD_LIBRARY_PATH=/opt/dim/linux:/opt/SFM/lib
export DIM_DNS_NODE LD_LIBRARY_PATH
# put here other common environment variable
# needed by many started processes
pkill tmSrv > /dev/null 2>&1
sleep 1
/opt/SFM/sbin/tmSrv -l 1 -p 1 -d online
```

With this script, process are started by default as user `online` (which must exist on the systems) but can also be stared as other users except user `root`. Messages are logged only to the DIM logger (the DIM service: `/<hostname>/logger/log`).

3.7. See also

`setuid(2), nice(1), setpriority(2), sched_setaffinity(2), sched_setscheduler(2), setsid(2), fork(2), umask(2), waitpid(2), chdir(2), clearenv(3), setenv(3), execv(3), kill(2), signal(7), glob(7), fnmatch(3), argz_add(3), dup2(2).`

4. The Task Manager command-line clients for Linux

4.1. tmStart

4.1.1. Synopsis

```
tmStart [-m hostname_pattern] TMS_Start_Command_String

tmStart [-h]
```

4.1.2. Description

Send the command string **`TMS_Start_Command_String`** to the CMD `/<HOSTNAME>/task_manager/start` of the TMSs of all the nodes whose hostname matches the POSIX.2 wildcard pattern **`hostname_pattern`**.

See 3.4.1 for a description of the **`TMS_Start_Command_String`** synopsis.

4.1.3. Options

`-h` Print the program usage and exit immediately.

`-m hostname_pattern`

(If present, it must be the first option). Send the command string **`TMS_Start_Command_String`** only to the nodes whose hostname matches the POSIX.2 wildcard pattern **`hostname_pattern`**. If not specified, signal is sent to all nodes on which the server `tmSrv` is running. **`hostname_pattern`** can contain '`*`', '`?`', character classes [...], ranges [0-4], and complementation. See `glob(7)`.

4.1.4. Environment

The program `tmStart` needs the two environment variables:

`DIM_DNS_NODE`

hostname.domain of DIM dns node.

LD_LIBRARY_PATH

Variable, in PATH format, which must contain the path of the shared libraries libdim.so, libSFMutils.so and libproc-3.2.3.so.

4.1.5. Warning

The wildcards in the command line must be escaped (by means of back-slash or by means of a couple of double quotation marks) in order to avoid the shell expansion.

4.1.6. Examples

```
tmStart /opt/SFM/tm/examples/counter
tmStart -m lxplus003 /opt/SFM/tm/examples/counter
tmStart -m "lxplus*" /opt/SFM/tm/examples/counter
tmStart -m lxplus\* /opt/SFM/tm/examples/counter
tmStart -m "lxplus00?" /opt/SFM/tm/examples/counter
tmStart -m lxplus00\? /opt/SFM/tm/examples/counter
tmStart -m "lxplus0[3-7]0" /opt/SFM/tm/examples/counter
tmStart -m "lxplus0[3-7]?" /opt/SFM/tm/examples/counter
tmStart -m lxplus003 -d -w /opt/SFM/tm/examples ./counter
tmStart -m lxplus003 -c -u myps -w /bin ps -e –f
tmStart -d /opt/SFM/tm/examples/counter
tmStart -d -s 1 -r 1 /opt/SFM/tm/examples/counter
tmStart -d -p -10 -n galli /opt/SFM/tm/examples/counter
tmStart -c -d –D LD_LIBRARY_PATH=/opt/dim/linux:/opt/SFM/lib
-D DIM_DNS_NODE=pcdom.fastwebnet.it
/opt/SFM/tm/examples/counter
tmStart –m lxplus003 –e –o /opt/SFM/tm/examples/counter
tmStart –m lxplus003 –a 0 –a 2 /opt/SFM/tm/examples/counter
```

4.2. tmLs

4.2.1. Synopsis

tmLs [-m hostname_pattern][utgid_pattern]

`tmLs [-h]`

4.2.2. Description

List the processes whose UTGID matches the POSIX.2 wildcard pattern **`utgid_pattern`** on all the nodes whose host-name matches the POSIX.2 wildcard pattern **`hostname_pattern`**. If **`utgid_pattern`** is not specified, it is set to "`*`".

4.2.3. Options

`-m hostname_pattern`

List only the processes on all the nodes whose host-name matches the POSIX.2 wildcard pattern **`hostname_pattern`**. If not specified, processes running on all nodes on which the server `tmSrv` is running are listed.

4.2.4. Environment

The program `tmLs` needs the two environment variables:

`DIM_DNS_NODE`

hostname.domain of DIM dns node.

`LD_LIBRARY_PATH`

Variable, in PATH format, which must contain the path of the shared libraries libdim.so, libSFMutils.so and libproc-3.2.3.so.

4.2.5. Technical note

`tmLs` command contacts **`tmSrv`** DIM service

`/<HOSTNAME>/task_manager/list`

which publishes the list of the UTGIDs of the processes running on the node which have UTGID variable set. **`list`** service on **`tmSrv`** lists only thread groups, not single threads to avoid duplicate UTGIDs in the list (all threads owning to the same thread group have the same UTGID set).

The list is updated not only periodically (every 10 seconds), but also whenever a TMS command is executed, one second after a TMS command has been executed and immediately after a process started by the current TMS has terminated.

4.2.6. Warning

The wildcards in the command line must be escaped (by means of back-slash or by means of a couple of double quotation marks) in order to avoid the shell expansion.

4.2.7. Examples

```
tmLs
tmLs counter_0
tmLs "count*"
tmLs count\*
tmLs "count*[2-5]"
tmLs -m lxplus003
tmLs -m "lxplus*"
tmLs -m lxplus\*
tmLs -m "lxplus0[3-7]?"
tmLs -m "lxplus*" "count*"
tmLs -m lxplus\* count\*
tmLs -m "lxplus0[3-7]?" "count*[2-5]"
```

4.2.8. See Also

`glob(7).`

4.3. tmKill

4.3.1. Synopsis

`tmKill [-m hostname_pattern] TMS_Kill_Command_String`

`tmKill [-h]`

4.3.2. Description

Send the command string **`TMS_Kill_Command_String`** to the CMD `/<HOSTNAME>/task_manager/kill` of the TMSs of all the nodes whose hostname matches the POSIX.2 wildcard pattern **`hostname_pattern`**.

See 3.4.2 for a description of the **`TMS_Kill_Command_String`** synopsis.

4.3.3. Options

`-m hostname_pattern`

(If present, it must be the first option). Send the command string **`TMS_Kill_Command_String`** only to the nodes whose hostname matches the POSIX.2 wildcard pattern **`hostname_pattern`**. If not specified, signal is sent to all nodes on which the server `tmSrv` is running. **`hostname_pattern`** can contain '`*`', '`?`', character classes [...], ranges [0-4], and complementation. See `glob(7)`.

4.3.4. Environment

The program `tmKill` needs the two environment variables:

`DIM_DNS_NODE`

hostname.domain of DIM dns node.

`LD_LIBRARY_PATH`

Variable, in PATH format, which must contain the path of the shared libraries libdim.so, libSFMutils.so and libproc-3.2.3.so.

4.3.5. Warning

The wildcards in the command line must be escaped (by means of back-slash or by means of a couple of double quotation marks) in order to avoid the shell expansion.

4.3.6. Examples

```
tmKill counter_0
tmKill "count*"
tmKill count\*
tmKill "count*[2-5]"
tmKill -m lxplus003 counter_0
tmKill -m "lxplus*" "count*"
tmKill -m lxplus\* count\*
tmKill -m "lxplus0[3-7]?" "count*[2-5]"
tmKill -s 2 counter_0
tmKill -s 2 "count*[2-5]"
tmKill -m lxplus003 -s 2 count_0
tmKill -m "lxplus*" -s 2 "count*"
tmKill "*"
tmKill \*
tmKill -s 2 "*"
tmKill -s 2 \*
tmKill -m "lxplus00[1357]" count_0
```

4.3.7. See Also

`glob(7), signal(7).`

4.4. tmStop

4.4.1. Synopsis

`tmStop [-m hostname_pattern] TMS_Stop_Command_String`

`tmStop [-h]`

4.4.2. Description

Send the command string **`TMS_Stop_Command_String`** to the CMD `/<HOSTNAME>/task_manager/stop` of the TMSs of all the nodes whose hostname matches the POSIX.2 wildcard pattern **`hostname_pattern`**.

4.4.3. Options

`-m hostname_pattern`

(If present, it must be the first option). Send the command string **`TMS_Stop_Command_String`** only to the nodes whose hostname matches the POSIX.2 wildcard pattern **`hostname_pattern`**. If not specified, signal is sent to all nodes on which the server `tmSrv` is running. **`hostname_pattern`** can contain '`*`', '`?`', character classes [...], ranges [0-4], and complementation. See `glob(7)`.

4.4.4. Environment

The program `tmStop` needs the two environment variables:

`DIM_DNS_NODE`

hostname.domain of DIM dns node.

`LD_LIBRARY_PATH`

Variable, in PATH format, which must contain the path of the shared libraries libdim.so, libSFMutils.so and libproc-3.2.3.so.

4.4.5. Warning

The wildcards in the command line must be escaped (by means of back-slash or by means of a couple of double quotation marks) in order to avoid the shell expansion.

4.4.6. Examples

```
tmStop counter_0
tmStop "count*"
tmStop count\*
tmStop "count*[2-5]"
tmStop -m lxplus003 counter_0
tmStop -m "lxplus*" "count*"
tmStop -m lxplus\* count\*
tmStop -m "lxplus0[3-7]?" "count*[2-5]"
tmStop -s 2 counter_0
tmStop -s 2 "count*[2-5]"
tmStop -m lxplus003 -s 2 counter_0
tmStop -m "lxplus*" -s 2 "count*"
tmStop -m lxplus003 -s 2 -d 4 counter_0
tmStop -m "lxplus*" -s 2 -d 4 "count*"
tmStop "*"
tmStop \*
tmStop -s 2 -d 4 "*"
tmStop -s 2 -d 4 \*
tmStop -m "lxplus00[1357]" counter_0
```

4.4.7. See Also

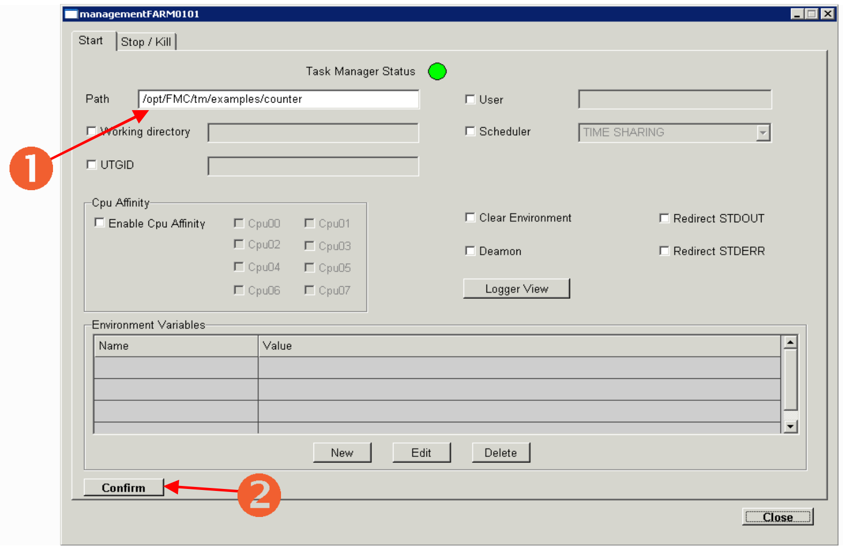
`glob(7), signal(7).`

5. The Task Manager PVSS client

The PVSS Task Manager Client is the lowest level graphical interface which is able to start, stop and list processes on servers which have the TMS running.

5.1. Start a process

The **simplest** command to start a process consists in typing in the field “**Path**” the path of the executable image to be run, and press the “**Confirm**” button, as shown in Figure 7.

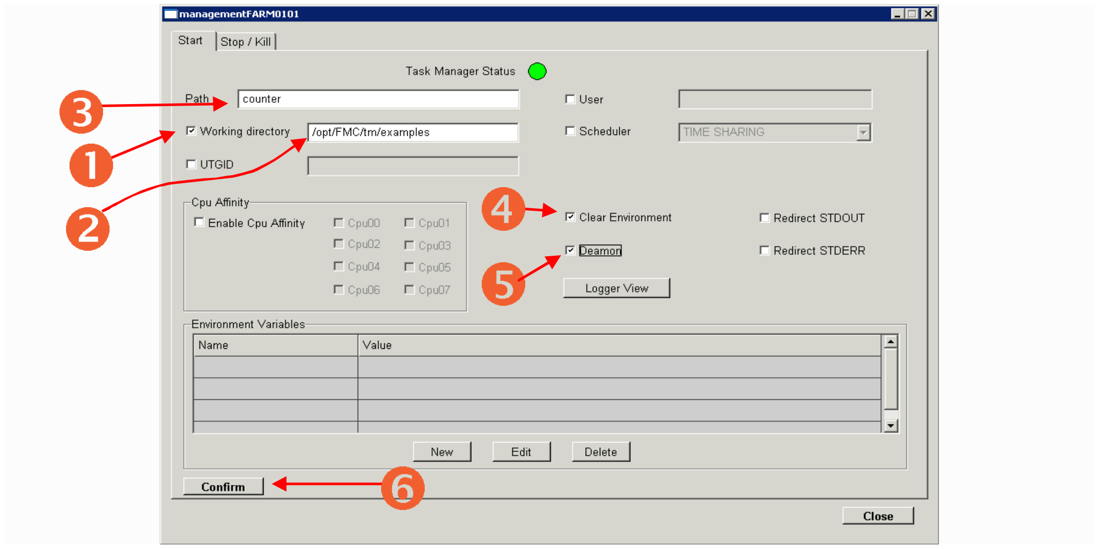


Path: /opt/FMC/tm/examples/counter

Figure 7. The simplest start command.

If the program to be started needs to open a file, specified as *relative* path, the process working directory must be set (by default the working directory is set to “/”), as shown in Figure 8, by setting the “**Working directory**” check-box and filling the “Working directory” text-field. In the example shown in Figure 8 also the “**Clear Environment**” check-box and the “**Daemon**” check-box are set: the first makes the process environment to be cleared (only the variables UTGID and PWD are set); the second makes the process running as daemon (process umask is reset, program is run in a new session as process group leader (**`setsid()`**)).

Starting a process as shown in Figure 7 and Figure 8 makes the Task Manager Server assign to the process an automatically generated UTGID string, which is build as `COMMAND+"_"+<instance_number>`. In the example, UTGID will be `counter_0` for the first instance, `counter_1` for the second instance and so on.



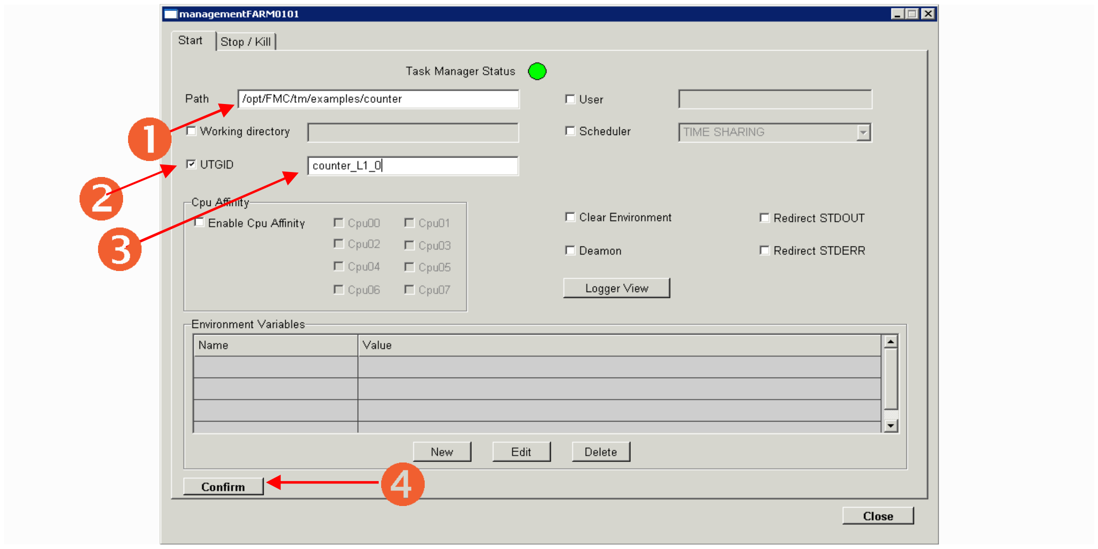
Path: counter
[✓] Working directory: /opt/FMC/tm/examples
[ ] UTGID
[ ] User
[ ] Scheduler: TIME SHARING
Cpu Affinity: [ ] Enable Cpu Affinity; [ ] Cpu00 [ ] Cpu01 [ ] Cpu02 [ ] Cpu03 [ ] Cpu04 [ ] Cpu05 [ ] Cpu06 [ ] Cpu07
[✓] Clear Environment
[ ] Redirect STDOUT
[✓] Deamon
[ ] Redirect STDERR
Environment Variables

Name	Value

Figure 8. Start command with working directory specification. The “Clear environment” check-box and “Daemon” check-box are also set.

Sometimes is desirable to assign a more mnemonic UTGID, for example to remember that the process is related to the L1 trigger. This can be achieved, as shown in Figure 9, by checking the “**UTGID**” check-box and filling the “UTGID” text-field.

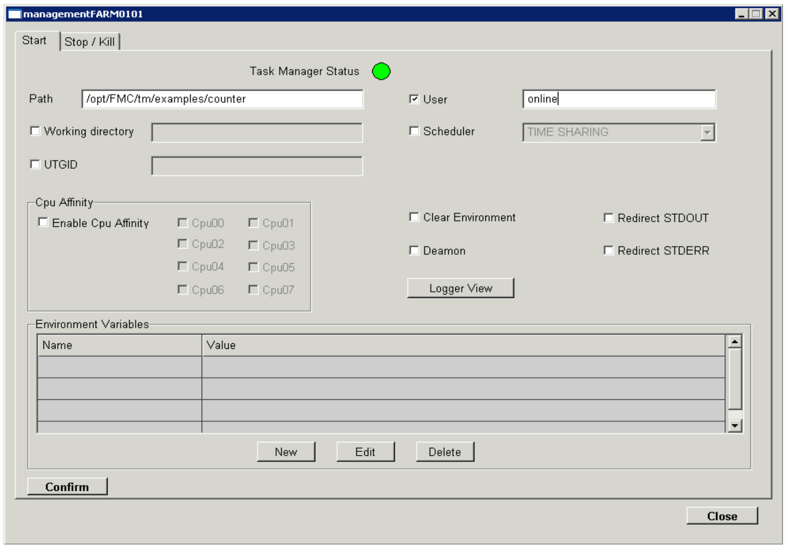
To start the process as a user different from the TMS default user (Figure 10), the “**User**” check-box must be set and the “User” text-field must be filled with the name of a user (which must be defined on the node on which the TMS is running). The choice of the user



Path: /opt/FMC/tm/examples/counter
[ ] Working directory
[✓] UTGID: counter_L1_0
[ ] User
[ ] Scheduler: TIME SHARING
Cpu Affinity: [ ] Enable Cpu Affinity; [ ] Cpu00 [ ] Cpu01 [ ] Cpu02 [ ] Cpu03 [ ] Cpu04 [ ] Cpu05 [ ] Cpu06 [ ] Cpu07
[ ] Clear Environment
[ ] Redirect STDOUT
[ ] Deamon
[ ] Redirect STDERR
Environment Variables

Name	Value

Figure 9. Start command with the UTGID specification.



Path /opt/FMC/tm/examples/counter

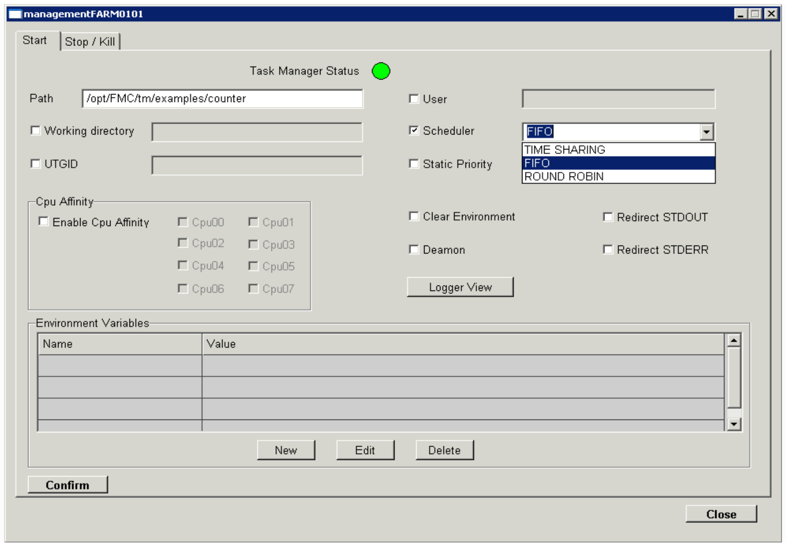
☑ User online

☐ Scheduler TIME SHARING

Figure 10. Start command with the user specification.

must be compatible with the policy defined in the TMS (see section 3.3), otherwise the process is not started and an error message is written to the logger.

The PVSS Task Manager client is also able to set the scheduler, the priority and the CPU affinity for the new process (the default is the TIME SHARING scheduler with 0 nice level and the process allowed to run on any CPU).

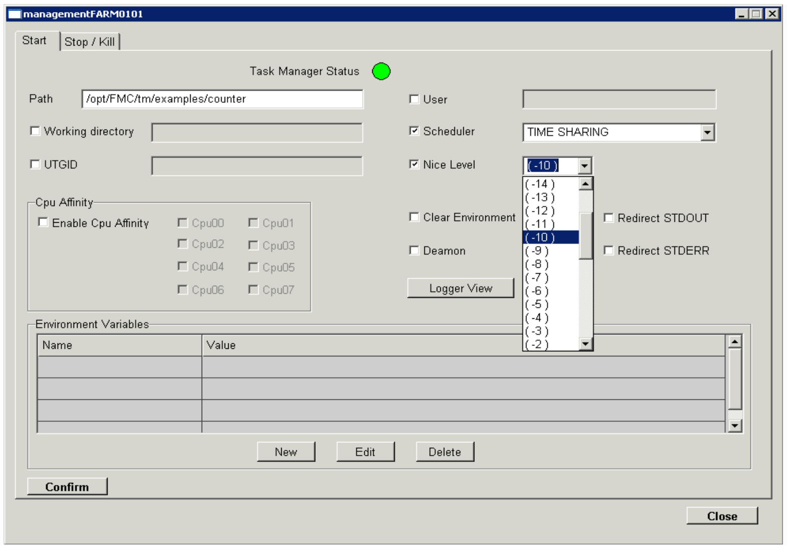


Path /opt/FMC/tm/examples/counter

☑ Scheduler FIFO

TIME SHARING
FIFO
ROUND ROBIN

Figure 11. Start command with the scheduler specification.



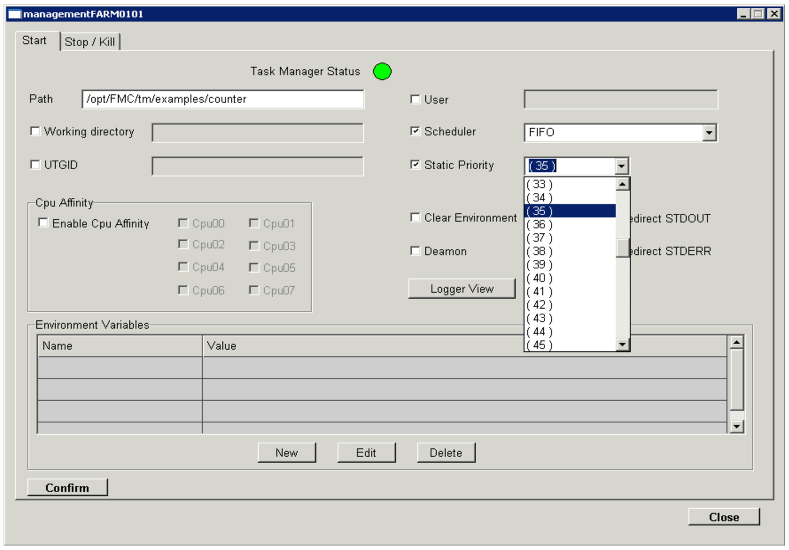
Path /opt/FMC/tm/examples/counter

Scheduler TIME SHARING

Nice Level (-10)

Figure 12. Start command with the TIME SHARING scheduler and the nice level specification.

In Figure 11 is shown the "**Scheduler**" check-box set, and the "Scheduler" combo-box opened. If the TIME SHARING scheduler is chosen, the "**Nice Level**" check-box can be set to activate the "Nice Level" combo-box to choose the process nice level, as shown in Figure 12. If the FIFO or the ROUND ROBIN scheduler is chosen, the "**Static Priority**"

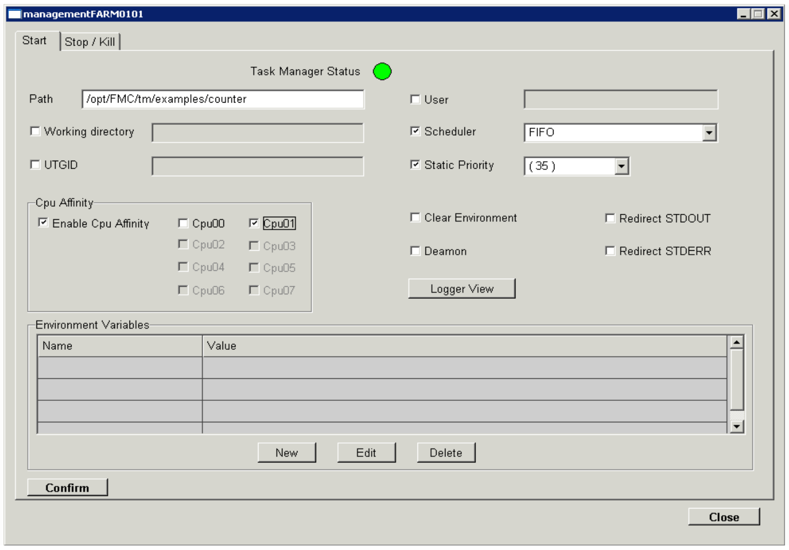


Path /opt/FMC/tm/examples/counter

Scheduler FIFO

Static Priority (35)

Figure 13. Start command with the FIFO scheduler and the static priority specification.



managementFARM0101

Start | Stop / Kill

Task Manager Status ●

Path	/opt/FMC/tm/examples/counter	☐ User	
☐ Working directory		☑ Scheduler	FIFO
☐ UTGID		☑ Static Priority	(35)

Cpu Affinity

☑ Enable Cpu Affinity	☐ Cpu00	☑ Cpu01
	☐ Cpu02	☐ Cpu03
	☐ Cpu04	☐ Cpu05
	☐ Cpu06	☐ Cpu07

☐ Clear Environment ☐ Redirect STDOUT
☐ Deamon ☐ Redirect STDERR
Logger View

Environment Variables

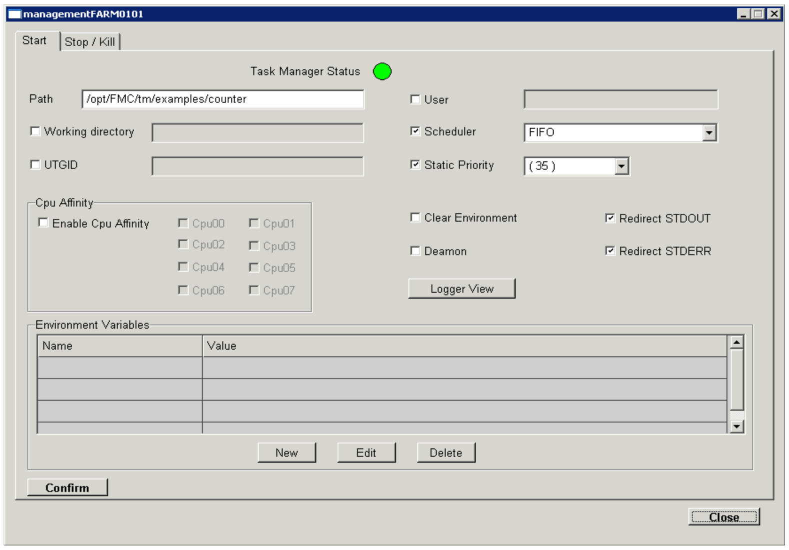
Name	Value

New Edit Delete
Confirm
Close

Figure 14. Start command with the FIFO scheduler, the static priority specification and the process-to-CPU affinity specification.

check-box can be set to activate the "Static Priority" combo-box to choose the process static priority, as shown in Figure 13.

To bind the process execution to one ore more specified CPUs, the process-to-CPU affinity mask can be set-up by checking the "**Enable Cpu Affinity**" check-box and then setting the check-boxes corresponding to the chosen CPU, as shown in Figure 14.



managementFARM0101

Start | Stop / Kill

Task Manager Status ●

Path	/opt/FMC/tm/examples/counter	☐ User	
☐ Working directory		☑ Scheduler	FIFO
☐ UTGID		☑ Static Priority	(35)

Cpu Affinity

☐ Enable Cpu Affinity	☐ Cpu00	☐ Cpu01
	☐ Cpu02	☐ Cpu03
	☐ Cpu04	☐ Cpu05
	☐ Cpu06	☐ Cpu07

☐ Clear Environment ☑ Redirect STDOUT
☐ Deamon ☑ Redirect STDERR
Logger View

Environment Variables

Name	Value

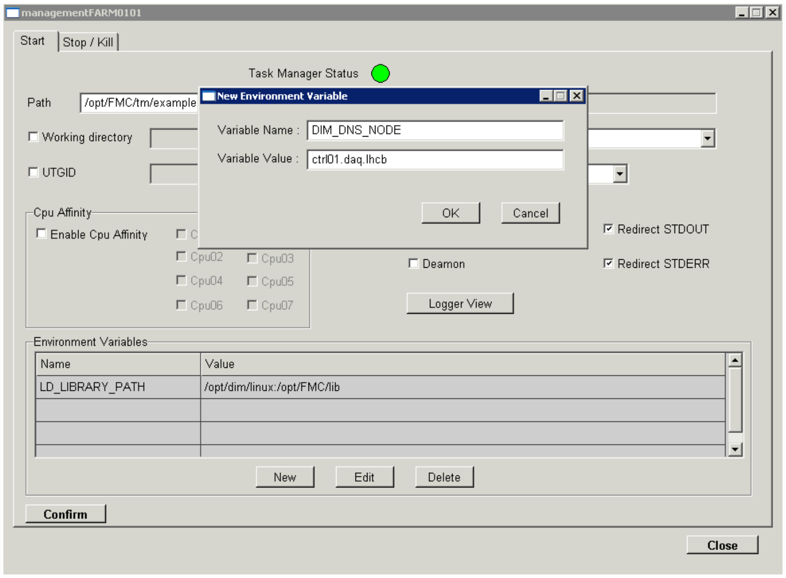
New Edit Delete
Confirm
Close

Figure 15. Start command with the redirection of the standard error and standard output of the started process to the Message Logger.

The standard output and the standard error of the started process, by default, are thrown in `/dev/null`. To redirect instead `stdout` and/or `stderr` of the started process to the **Message Logger** (`/tmp/logSrv.fifo`), the "**Redirect STDOUT**" and/or "**Redirect STDERR**" check-boxes can be set, as shown in Figure 15.

A further control regards the process environment variables. We have already seen that the "Clear Environment" check-box can be used to choose whether the started process must inherit or not the environment variables set for the Task Manager process itself. If a group of environment variables is needed by many processes to be started, the easiest solution is to add these variables to the `tmSrv` start-up script and to make the started process inherit them.

It is possible, however, to set additional environment variables, specific for the started process, by using the 3 buttons in the "**Environment Variables**" frame. To add a new environment variable for the process to be started, you can press the "**New**" button, fill the two text-fields in the pop-up window (Variable Name and Variable Value) and then press the "OK" button, as you can see in Figure 16. As a result you will se the new variable inserted in the "Environment Variables" list (Figure 17). All the variables set in the "Environment Variables" list will be passed to the started process. The "**Edit**" button can be then used to modify an environment variable previously set; the "**Delete**" button can be used to remove an environment variable previously set.



Path /opt/FMC/tm/example

Variable Name : DIM_DNS_NODE

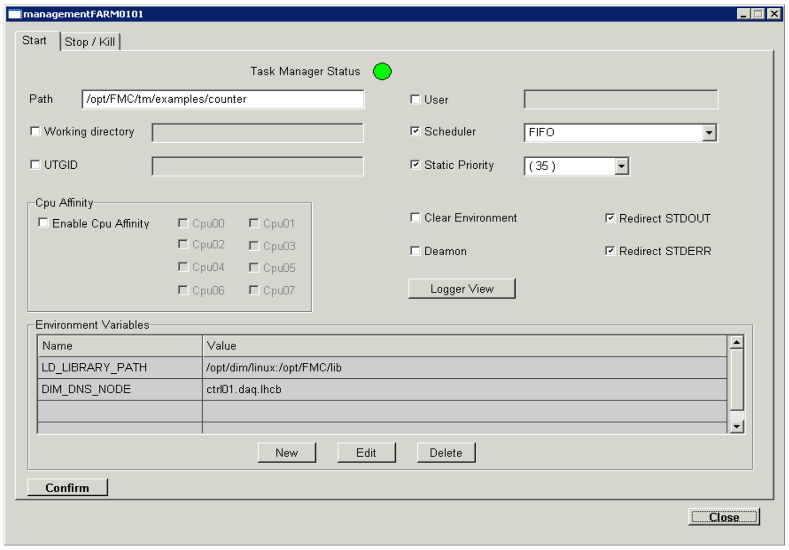
Variable Value : ctrl01.daq.lhcb

☑ Redirect STDOUT

☑ Redirect STDERR

Name	Value
LD_LIBRARY_PATH	/opt/dim/linux:/opt/FMC/lib

Figure 16. Start command with additional environment variables. The "New" button opens the pop-up window to set a new variable.



Path: /opt/FMC/tm/examples/counter

☐ Working directory
☐ UTGID
☐ User
☑ Scheduler: FIFO
☑ Static Priority: (35)

Cpu Affinity
☐ Enable Cpu Affinity
☐ Cpu00 ☐ Cpu01
☐ Cpu02 ☐ Cpu03
☐ Cpu04 ☐ Cpu05
☐ Cpu06 ☐ Cpu07

☐ Clear Environment
☐ Deamon
☑ Redirect STDOUT
☑ Redirect STDERR

Environment Variables

Name	Value
LD_LIBRARY_PATH	/opt/dim/linux:/opt/FMC/lib
DIM_DNS_NODE	ctrl01.daq.lhcb

Figure 17. Start command with additional environment variables. The new variable has been added to the environment variable list.

5.2. Send a signal to a process

To send a signal to a process, in the Task Manager panel the “**Stop/Kill**” tabbed pane must be chosen, the “**kill**” check-box (in the “**Action Type**” frame) must be set, and the process (or the processes) which must receive the signal must be chosen from the list.

If a signal different from SIGTERM (signal number 15) must be sent, the “**Signal**” check-box must be set and the appropriate signal must be chosen from the “Signal” combo-box. Finally the “**Kill Process**” button must be pressed (Figure 18).

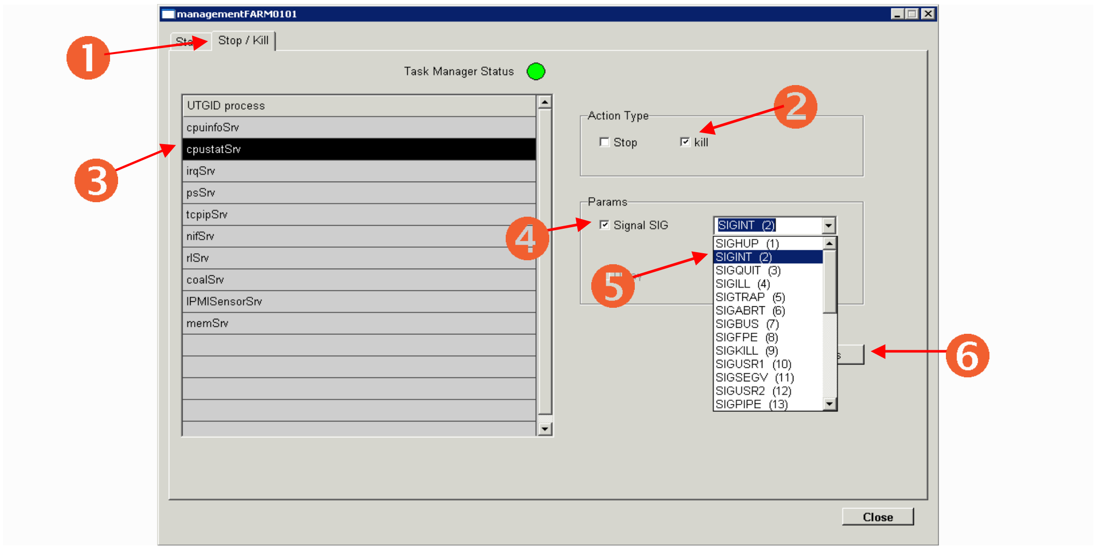


Figure 18. Sending a signal to a process (kill command).

5.3. Stop a process

To stop a process, in the Task Manager panel the "**Stop/Kill**" tabbed pane must be chosen, the "**stop**" check-box (in the "**Action Type**" frame) must be set, and the process (or the processes) which must receive the signal must be chosen from the list.

If a signal different from SIGTERM (signal number 15) must be used as first signal to stop the process, the "**Signal**" check-box must be set and the appropriate signal must be chosen from the "Signal" combo-box. Finally the "**Stop Process**" button must be pressed (Figure 19). This way the chosen signal is sent to the chosen processes and the deferred execution of finishOffPs() function is scheduled. After one second, finishOffPs() function controls whether the process (or the processes) are already dead, otherwise a second signal (a SIGKILL signal this time) is sent to the processes which are still alive.

The delay of execution of finishOffPs() function (default one second) can be changed by setting the "**Delay**" check-box and writing the appropriate delay (in seconds) in the "Delay" text-field (Figure 20).

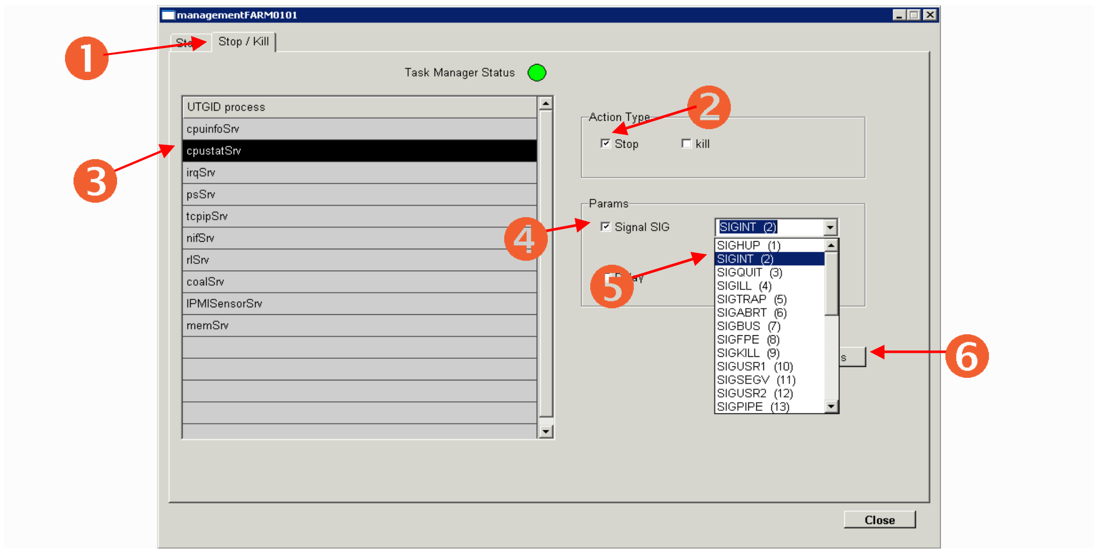


Figure 19. Stopping a process (stop command).

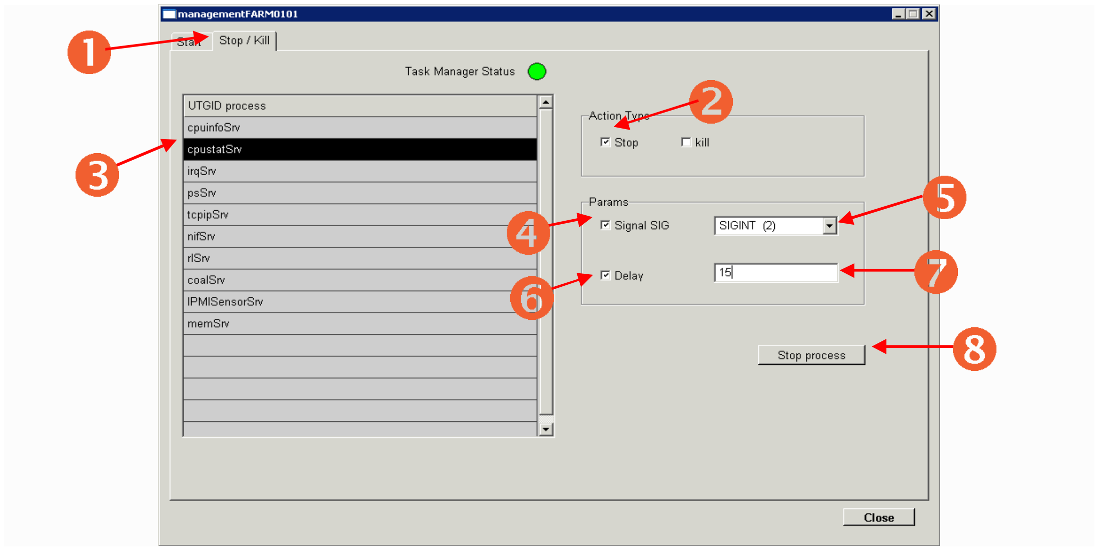


Figure 20. Stopping a process with the SIGKILL delay specification.

6. References

[1] C. Gaspar, DIM, Distributed Information Management System: see URL http://dim.web.cern.ch/dim/.

[2] C. Gaspar, PVSS - DIM Integration: see URL http://clara.home.cern.ch/clara/fw/FwDim.html.

[3] LHC Experiments Joint COntrol Project: see URL http://itco.web.cern.ch/itco/Projects-Services/JCOP/welcome.html.

[4] PVSS Service: see URL http://itcobe.web.cern.ch/itcobe/Services/Pvss/.