The Message Logger for the LHCb On-Line Farm

LHCb Technical Note

Issue:	1
Revision:	1
Reference:	LHCb 2005-050 DAQ
Created:	4 Aug. 2005
Last modified:	23 Aug. 2005

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

Abstract

The Message Logger is a utility which provide a logger facility for the processes running on the nodes of the on-line farm. It can also be used to collect the processes' `stdout/stderr`. The Message Logger can be exploited using two different policies: either as a *no-drop* logger facility (messages cannot be lost, but a write to the logger facility blocks in case of full-buffer condition, due e.g. to a network congestion) or as a *congestion-proof* logger facility (a write to the logger facility never locks even in case of network congestion, but, in this case, messages are dropped). The Message Logger package includes a Linux DIM server (`logSrv`), a Linux terminal/command-line DIM client (`logViewer`) and a PVSS DIM client.

Document Status Sheet

Table 1 Document Status Sheet

1. Document Title: The Message Logger for the LHCb On-Line Farm			
2. Document Reference Number: LHCb 2005-050 DAQ			
3. Issue	**4. Revision**	**5. Date**	**6. Reason for change**
Draft	0	4 Aug. 2005	First version
1	1	23 Aug. 2005	First released version

Table of Contents

List of Figures

1. Requirements

During the operation of the on-line farm, several processes running on the farm nodes may need to send messages either for **debugging purpose** or to **signal some critic or error conditions**.

These messages cannot be sent to a terminal because the processes run in background, not bound to a particular terminal and often as daemons (process group leader).

These messages could be written to one or several files, but, for a diskless farm, this would mean to keep several NFS-like connections open. Moreover these files will increase indefinitely their size, so they will need to be periodically trimmed.

Processes could also send directly the message through the network, e.g. publishing a DIM service. This choice, however, has the drawback to have no buffer between the processes and the network and to impose a blocking write mode to the processes: in case of congested network, the application may block trying to send messages.

A better choice would be a client-server application (Figure 1), in which:

- the server (running on each farm node) collects in a small buffer the messages from all the processes running on its node (avoiding interleaving them) and

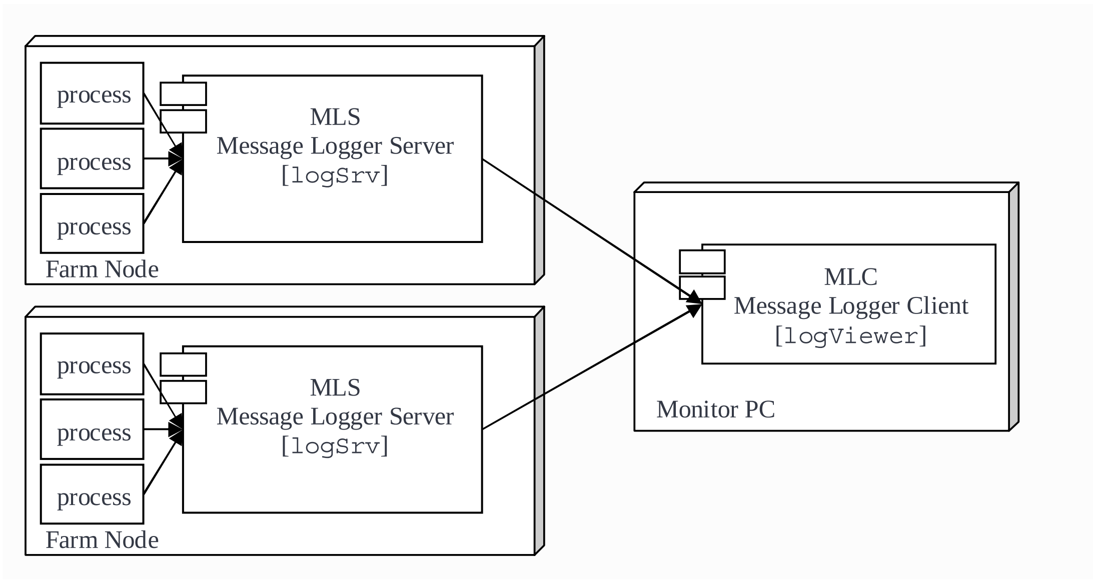


Figure 1. The Message Logger deployment.

send them asynchronously through the network;

- the client (running on another PC) collects the messages from on one or more specified nodes of the farm (even from a whole sub-farm or from the whole farm), merges them together (avoiding interleaving them) and displays them in a terminal-like window (which could be a true terminal, a terminal emulator like xterm or gnome-terminal or a PVSS window).

Two different working policies can be required to this kind of utility:

- *No-drop policy*: preservation of all messages is privileged with respect to the ability to run without blocking in network congestion conditions. Messages cannot be lost, but a write to the logger facility blocks in case of a buffer full condition due e.g. to a network congestion.

- *Congestion-proof policy*: ability to run without blocking, even in network congestion condition, is privileged with respect to message preservation. A write to the logger facility never locks, even in case of network congestion, but, in this case, messages are dropped.

The congestion-proof policy is preferred for the LHCb on-line farm. As a matter of fact, in critic farm conditions (e.g. due to a temporary network congestion), we prefer to loose messages rather than to have all the control and monitor processes blocked, waiting to send warning messages.

2. Implementation

The Message Logger System is implemented using DIM (Distributed Information Management System) as network communication layer [1] and a POSIX FIFO as a buffer and a method for inter-process communication (Figure 2).

2.1. DIM

DIM has client/server architecture and uses a Name Server mechanism to publish/subscribe services (Figure 2).

Each farm node runs a Message Logger Server (MLS, whose executable name is `logSrv`), which registers a service (`/<HOSTNAME>/logger/log`) with the DIM Name Server and makes it available to the logger client (s2 in Figure 2).

The Message Logger Client (MLC, whose command-line executable is named `logViewer`), asks the DIM Name Server which server makes the log services available (c1 in Figure 2); once got the answer (c2) the MLC contacts directly the servers (c3) to subscribe to the log services (to bring itself up-to-date about the new messages, r3 in Figure 2).

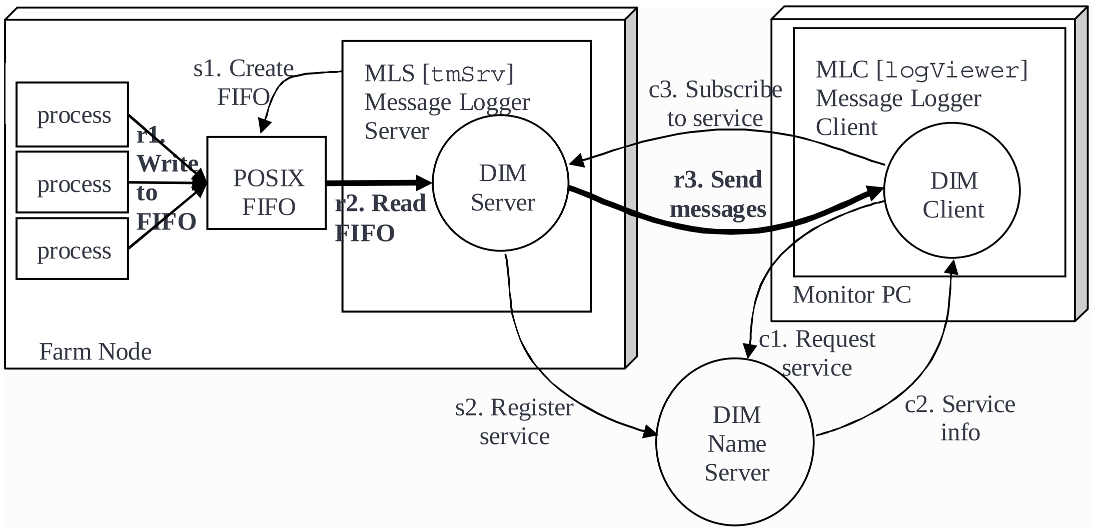


Figure 2. The Message Logger’s collaboration diagram. s1-s2: Server start-up; c1-c3: client start-up; r1-r3 message flow.

2.2. The POSIX FIFO

A POSIX.1 *unnamed pipe* is a sequential, uni-directional, communication stream between two processes, managed by the Operating System kernel.

A *POSIX.1 FIFO* (alias *named pipe*) is an extension to the classical unnamed pipe concept, which allows bidirectional, multipoint communications between unrelated processes.

A *special file* is always associated with a POSIX.1 FIFO. The FIFO special file is created with the `mkfifo(3)` library call and can be opened by multiple processes for reading or writing. When processes are exchanging data via the FIFO, the kernel passes all data internally without writing them to the file system. Thus, the FIFO special file has no contents on the file system; the file system entry merely serves as a reference point so that processes can access the pipe using a name in the file system.

2.3. Opening a POSIX.1 FIFO

In Message Logger System, the Message Logger Server opens the FIFO for *reading*, while the processes which need to send messages open the FIFO for *writing*. POSIX standard foresees two modes for FIFO opening: blocking and non-blocking.

Opening a FIFO for *reading only* (`O_RDONLY`), in *blocking mode* (flag `O_NONBLOCK` clear) will block the calling process until another process opens the FIFO for writing; in *non-blocking mode* (flag `O_NONBLOCK` set) it will always return without delay, even when there is no process writing the FIFO.

Opening a FIFO for *writing only* (`O_WRONLY`), in *blocking mode* (flag `O_NONBLOCK` clear) will block the calling process until another process opens the FIFO for reading; in *non-blocking mode* (flag `O_NONBLOCK` set) it will return an error (`ENXIO`) if no process currently has the FIFO open for reading.

Opening a FIFO for *reading and writing* (`O_RDWR`) has undefined behaviour in POSIX.1. In Linux implementation [2], open will never block and never return an error (since the process is supposed to be able at least talk to itself).

In the Message Logger System:

- the **MLS** opens the FIFO for reading only, in non-blocking mode (`O_RDONLY|O_NONBLOCK`),
- the **application** which needs to send messages can open the FIFO in 2 modes:
    - Open the FIFO for writing only in blocking mode (`O_WRONLY |O_APPEND`) to achieve *no-drop* behaviour.

- Open the FIFO for reading and writing in non-blocking mode (`O_RDWR|O_NONBLOCK|O_APPEND`) to achieve *congestion-proof* behaviour.

2.4. No-drop and congestion-proof behaviour

An application which needs to send messages to the Message Logger Server has to open the FIFO and write messages to it (the LHCb Task Manager can do the work for you, and then redirect to the FIFO the started process's `stdout/stderr`).

If the application opens the FIFO for writing only in blocking mode (`O_WRONLY|O_APPEND`) then the *no-drop* behaviour is achieved. If the application opens the FIFO for reading and writing in non-blocking mode (`O_RDWR |O_NONBLOCK|O_APPEND`) then the *congestion-proof* behaviour is achieved

We emphasize here that, using the *congestion-proof* policy in a network **congestion condition**, the **messages are dropped by the FIFO, non by the network** (as would be achieved by moving messages through network using UDP). This way, in a congestion condition, **no additional message traffic is sent through the network**: messages are dropped before they reach the network.

2.5. FIFO size

FIFOs and pipes implementation has changed in Linux kernel 2.6.11.

Before kernel 2.6.11 the pipe and FIFO data structure was a circular list which fitted exactly in one memory page. The size of a FIFO buffer allocated by the Linux operating system was therefore exactly the size of a memory page (`PIPE_SIZE` = `PAGE_SIZE` = = 1 << 12 = 4 KiB, defined in `/usr/include/linux/pipe_fs_i.h`).

Starting with kernel 2.6.11, the pipe data structure has became a circular list of memory pages (`PIPE_BUFFERS` = 16 memory pages, defined in `/usr/include/linux/pipe_fs_i.h` as well), for a total of 64 KiB.

2.6. FIFO write atomicity

The maximum number of bytes which is possible to put in a FIFO in a single *atomic write operation* is set by the parameter `PIPE_BUF`, which is set to 4 KiB in the header file `/usr/include/linux/pipe_fs_i.h`.

If the number of bytes written to a pipe exceeds the atomic limit for a single operation, and multiple processes are writing to the pipe, the data will be "interleaved" or "chunked". In other words, one process may insert data into the pipeline between the writes of another.

Thus the **size of the messages sent to the logger must be limited to 4096 characters** (about 51 rows on a standard, 80 columns, terminal), **in order to avoid message interleaving**.

2.7. Recognizing the Severity Level

If a message is sent from an application to the MLS using the provided library call `msgSend(fName,severity,msg)`, then a header string with the format "`MMMdd-hhmmss[SEVER]host:`" (e.g.: "`Aug10-142541[INFO]lhcbcn2:`") is pre-pended by `msgSend()` to the message sent to the MLS.

Otherwise (if the square brackets [] of the header are not found in the message by the MLS, that happens, for example, if the message is sent to the MLS by redirecting `stdout/stderr`), the MLS itself adds a header, trying to recognize the severity level by looking at the message for the strings (case-insensitive): "error" and "denied": if such strings are found then security level is set to `ERROR`, else security level is set to `DEBUG`.

3. The Message Logger Server (MLS)

The MLS, whose executable name is `logSrv`, have **to run on each node of the farm**. It is recommended to start the MLS from the `init` process, using the `respawn inittab` option in order to insure that it is always alive.

3.1. Synopsis

`logSrv [-p fifo_path][-s srv_name]`

`logSrv [-h]`

3.2. Description

Starts the MLS on the node. The MLS first creates a POSIX FIFO, opens the FIFO for reading and registers the log service with the DIM Name Server; then continuously reads messages from the FIFO, add them a header if it is not already present (see 2.6), and publishes them using DIM, blocking when no new message arrives.

The default FIFO path name and the default DIM service name are, respectively: **`/tmp/logSrv.fifo`** and **`/<HOSTNAME>/logger/log`**. These defaults can be changed using **`-p`** and **`-s`** command line options.

The change in FIFO path name and in DIM service name can be useful to start more than one logger on the same farm node (e.g. a logger for a certain group of processes and another logger for another group of processes). In this case both the `fifo_path` and the `srvc_name` must be different for the two loggers.

3.3. Command line options

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

`-p fifo_path`

Use **`fifo_path`** as the FIFO path name. Default FIFO path name: `/tmp/logSrv.fifo`.

-s srvc_name

Use **/<HOSTNAME>/<srvc_name>/log** as DIM service name. Default DIM service name: /<HOSTNAME>/logger/log.

3.4. Environment

The program logSrv 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 and libSFMutils.so.

3.5. Examples

The Message Logger Server logSrv can be started using the inittab, writing in /etc/inittab an entry like:

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

where <Id> is a unique sequence of 1-4 characters which identifies an entry in the inittab, <run_level> lists the run-levels for which the TMS have to run, and startLogSrv.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
pkill logSrv > /dev/null 2>&1
sleep 1
/opt/SFM/sbin/logSrv
```

With this script, TMS is started, using /tmp/logSrv.fifo as FIFO path and /<HOSTNAME>/logger/log as DIM service name.

To run a second MLS on the same node, the following script can be used:

```
#!/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
```

```
/opt/SFM/sbin/logSrv -p /tmp/logSrv-2.fifo -s logger-2
```

With this script, a TMS is started, using `/tmp/logSrv-2.fifo` as FIFO path and `/<HOSTNAME>/logger-2/log` as DIM service name, and can therefore live together with the TMS started with the previous script.

3.6. See also

`fifo(4)`, `mkfifo(3)`, `open(2)`, `/usr/src/linux/fs/fifo.c`.

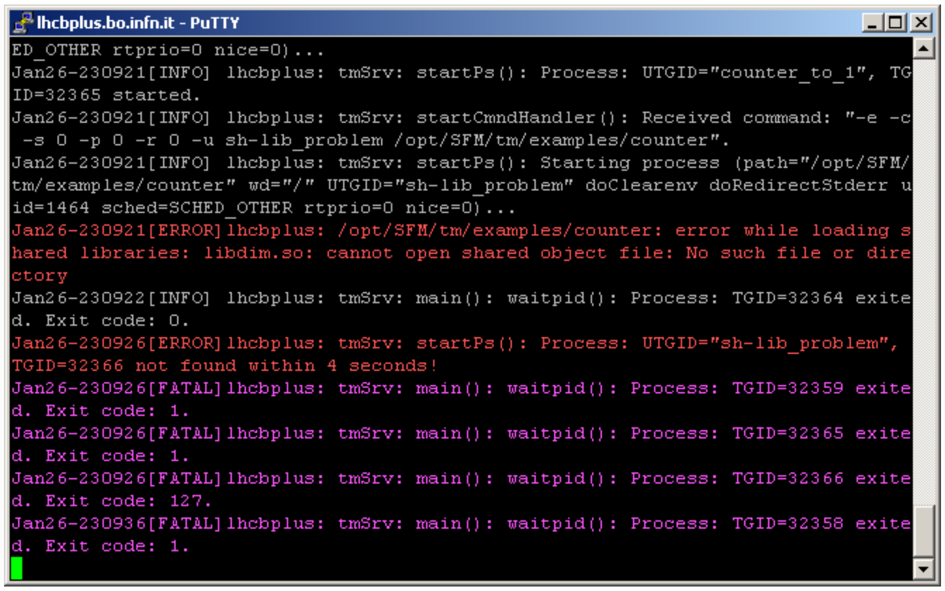

```
ED_OTHER rtprio=0 nice=0)...
Jan26-230921[INFO] lhcbplus: tmSrv: startPs(): Process: UTGID="counter_to_1", TG
ID=32365 started.
Jan26-230921[INFO] lhcbplus: tmSrv: startCmndHandler(): Received command: "-e -c
 -s 0 -p 0 -r 0 -u sh-lib_problem /opt/SFM/tm/examples/counter".
Jan26-230921[INFO] lhcbplus: tmSrv: startPs(): Starting process (path="/opt/SFM/
tm/examples/counter" wd="/" UTGID="sh-lib_problem" doClearenv doRedirectStderr u
id=1464 sched=SCHED_OTHER rtprio=0 nice=0)...
Jan26-230921[ERROR]lhcbplus: /opt/SFM/tm/examples/counter: error while loading s
hared libraries: libdim.so: cannot open shared object file: No such file or dire
ctory
Jan26-230922[INFO] lhcbplus: tmSrv: main(): waitpid(): Process: TGID=32364 exite
d. Exit code: 0.
Jan26-230926[ERROR]lhcbplus: tmSrv: startPs(): Process: UTGID="sh-lib_problem",
TGID=32366 not found within 4 seconds!
Jan26-230926[FATAL]lhcbplus: tmSrv: main(): waitpid(): Process: TGID=32359 exite
d. Exit code: 1.
Jan26-230926[FATAL]lhcbplus: tmSrv: main(): waitpid(): Process: TGID=32365 exite
d. Exit code: 1.
Jan26-230926[FATAL]lhcbplus: tmSrv: main(): waitpid(): Process: TGID=32366 exite
d. Exit code: 127.
Jan26-230936[FATAL]lhcbplus: tmSrv: main(): waitpid(): Process: TGID=32358 exite
d. Exit code: 1.
```

Figure 3. Screen dump of the terminal version of the Message Logger Client [`logViewer`].

4. The Command-Line Message Logger Client (MLC)

4.1. Synopsis

`logViewer [-b][-l level][-s srvc_name]`

`logViewer [-h]`

4.2. Description

Starts the MLC on the node. The MLC first tries to discover one or more DIM logger services by looking into the DIM Name Server for services named (by default) **`*/logger/log`** (where the asterisk is interpreted as wildcard). Then the MLC subscribes to them and print to the standard terminal the messages received from the subscribed logger services. The number of the subscribed logged services can be *limited* by using the **`-s`** command line switch, which accepts the POSIX.2 standard wildcard ('*', '?', character classes and ranges).

The MLC recognizes the *severity level* of the logged messages and can print only messages with a severity level greater than or equal to a chosen level, by means of the **`-l`** command line switch. Severity of the messages is recognized by a label (`[DEBUG]`, `[INFO]`, `[WARN]`, `[ERROR]`, `[FATAL]`) included in the message strings.

As a default, the MLC prints messaged using different *colours*, corresponding to the different *severity level*, by using the *ANSI escape sequences*. The **`-b`** command line switch can be used to print in black and white (useful for terminals which do not support the ANSI escape sequences).

4.3. Options

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

`-b` Print the messages in black and white. Useful for terminal which do not support the ANSI escape sequences.

`-s srvc_name`

Subscribe only to logger services matching the POSIX.2 wildcard expression **`srvc_name`** (this can be useful to limit the number of messages printed on a terminal). Recognized expressions include '*', '?', character classes, ranges, complementation. Wildcard must be escaped (using either a backslash for the single wildcard or a couple of double quotation marks for the whole string, see examples). Default wildcard expression: **`*/logger/log`**.

`-l level`

Print only logged messages with severity level greater than or equal to **`level`**. The severity level can be specified either as an integer number (in the range 0…5) or as a string (`ALL`, `DEBUG`, `INFO`, `WARN`, `ERROR`, `FATAL`).

4.4. Environment

The program `logViewer` 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 library libdim.so.

4.5. Examples

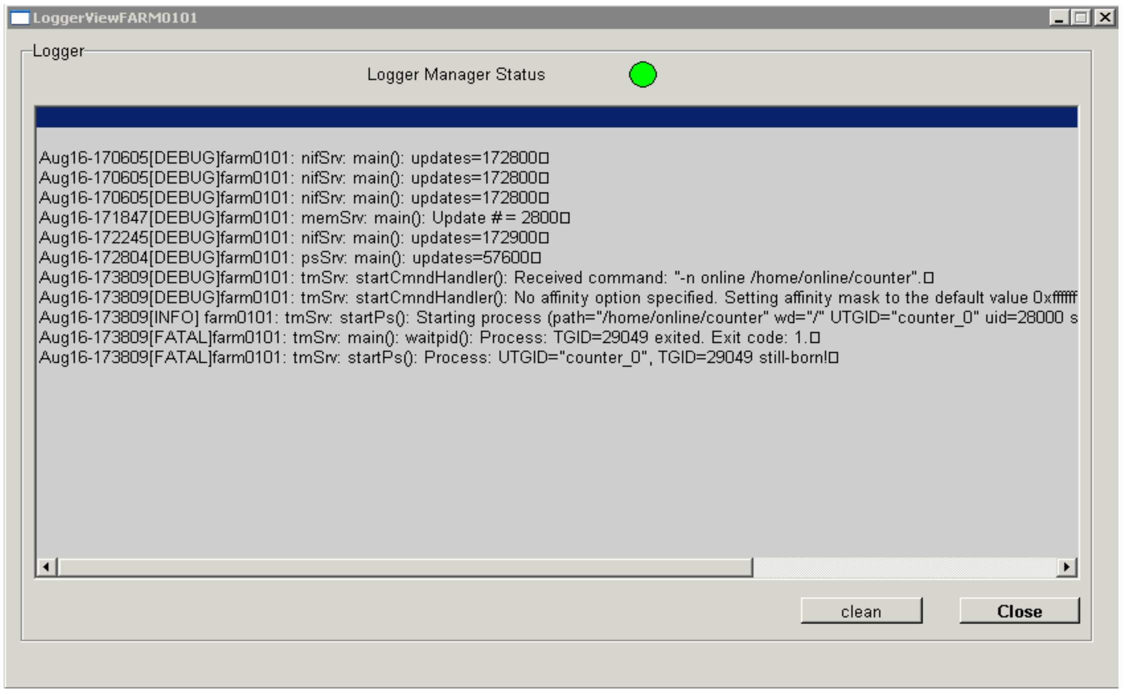
```
logViewer
logViewer -b
logViewer -l WARN
logViewer -l 3
logViewer -l INFO -b -s "*logger-2/log"
logViewer -l INFO -b -s \*logger-2/log
logViewer -l WARN -s "/LXPLUS00[3-7]/logger/log"
```

4.6. See also

`glob(7),fnmatch(3).`

5. The PVSS Message Logger Client (MLC)

Has the same functionalities of the Command-Line Message Logger Client but uses a PVSS graphic window instead of a terminal or a terminal emulator.



```
Aug16-170605[DEBUG]farm0101: nifSrv: main(): updates=172800□
Aug16-170605[DEBUG]farm0101: nifSrv: main(): updates=172800□
Aug16-170605[DEBUG]farm0101: nifSrv: main(): updates=172800□
Aug16-171847[DEBUG]farm0101: memSrv: main(): Update #= 2800□
Aug16-172245[DEBUG]farm0101: nifSrv: main(): updates=172900□
Aug16-172804[DEBUG]farm0101: psSrv: main(): updates=57600□
Aug16-173809[DEBUG]farm0101: tmSrv: startCmndHandler(): Received command: "-n online /home/online/counter".□
Aug16-173809[DEBUG]farm0101: tmSrv: startCmndHandler(): No affinity option specified. Setting affinity mask to the default value 0xffffff
Aug16-173809[INFO] farm0101: tmSrv: startPs(): Starting process (path="/home/online/counter" wd="/" UTGID="counter_0" uid=28000 s
Aug16-173809[FATAL]farm0101: tmSrv: main(): waitpid(): Process: TGID=29049 exited. Exit code: 1.□
Aug16-173809[FATAL]farm0101: tmSrv: startPs(): Process: UTGID="counter_0", TGID=29049 still-born!□
```

Figure 4. Screen dump of the PVSS version of the Message Logger Client.

6. Writing to the logger

In the following sections we will show three different way for an application to send data to the MLS.

6.1. Writing to the Logger by redirecting the stdout/stderr

The simplest way for a process to use the Message Logger consists in redirecting the standard error and/or the standard output of the process to the logger FIFO (`/tmp/logSrv.fifo` by default). This allows **using the Message Logger with the existing applications without modifying them**.

Redirection can be achieved both by starting the process by the shell command-line and by starting the process using the LHCb Task Manager [4].

The drawback of this method is the lack of control on the severity level and logging policy. If the message contains the word (case-insensitive) “error” or “denied” the severity level is set to `ERROR`, otherwise it is set to `DEBUG`; the logging policy is set to *no-drop* by the shell command line and is set to *congestion-free* by the Task Manager (which opens the FIFO using the `O_RDWR|O_NONBLOCK` flag).

For example, by using the **bash shell** command-line, you can type:

```
[user@host home]$ myapplication >/tmp/logSrv.fifo &
```

to redirect standard output to the MLS,

```
[user@host home]$ myapplication 2>/tmp/logSrv.fifo &
```

to redirect standard error to the MLS and

```
[user@host home]$ myapplication >/tmp/logSrv.fifo 2>&1 &
```

to redirect both standard output and standard error to the MLS.

By using the **LHCb Task Manager** the standard output and standard error can be redirected to the default logger FIFO (`/tmp/logSrv.fifo`) using the `-o` and/or the `-e` options. For example, by using the LHCb Task Manager from the command-line client, you can type:

```
[user@host home]$ tmStart -o myapplication
```

to redirect standard output to the MLS,

```
[user@host home]$ tmStart –e myapplication
```

to redirect standard error to the MLS and

```
[user@host home]$ tmStart –o –e myapplication
```

to redirect both standard output and standard error to the MLS.

Messages can then be sent to the Message Logger Server using the standard library calls `printf(…)` or `fprintf(stderr,…)`.

6.2. Writing to the Logger by opening the FIFO

A process can open the logger FIFO (`/tmp/logSrv.fifo`) and write messages to it.

A FIFO can be opened in blocking mode to achieve a **no-drop** logging policy. An example, which uses the buffered streams (`fprintf(3)`), is the following:

```
#include <stdio.h>
int main()
{
  FILE *fifoFP;
  char message[4096]=”my message”;
  fifoFP=fopen("/tmp/logSrv.fifo","a");
  setlinebuf(fifoFP); /* to avoid message interleave */
  fprintf(fifoFP,"%s",message);
  return 0;
}
```

Another example, which uses the low-level `write(2)`, is the following:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>
int main()
{
  int fifoFD;
  char message[4096]=”my message”;
  fifoFD=open("/tmp/logSrv.fifo",O_WRONLY|O_APPEND);
  write(fifoFD,message,1+strlen(message));
  return 0;
}
```

A further example, in which the process redirects the standard output from itself, is the following:

```
#include <stdio.h>
int main()
{
  char message[4096]=”my message”;
  freopen("/tmp/logSrv.fifo","a",stdout);
  setlinebuf(stdout); /* to avoid message interleave */
  printf("%s",message);
  return 0;
}
```

A FIFO can also be opened in non-blocking mode to achieve a **congestion-free** logging policy. An example, which uses the buffered streams (`fprintf(3)`), is the following:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
int main()
{
  int fifoFD;
  FILE *fifoFP;
  char message[4096]=”my message”;
  fifoFD=open("/tmp/logSrv.fifo",O_RDWR|O_NONBLOCK|
                               O_APPEND);
  fifoFP=fdopen(fifoFD,"a");
  setlinebuf(fifoFP); /* to avoid message interleave */
  fprintf(fifoFP,"%s",message);
  return 0;
}
```

Another example, which uses the low-level `write(2)`, is the following:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
int main()
{
  int fifoFD;
  char message[4096]=”my message”;
  fifoFD=open("/tmp/logSrv.fifo",O_RDWR|O_NONBLOCK|
                               O_APPEND);
  write(fifoFD,message,1+strlen(message));
  return 0;
}
```

A further example, in which the process redirects the standard output from itself, is the following:

```
#include <sys/types.h>
#include <sys/stat.h>
```

```
#include <fcntl.h>
#include <stdio.h>
int main()
{
  int fifoFD;
  char message[4096]=”my message”;
  fifoFD=open("/tmp/logSrv.fifo",O_RDWR|O_NONBLOCK|
                                 O_APPEND);
  dup2(fifoFD,STDOUT_FILENO);
  printf("%s",message);
  return 0;
}
```

6.3. Writing to the Logger by using the Message Utility API

A process can send messages to the MLS using the API defined in the header `msgUtils.h` and implemented in the shared library `libSFMutils-0.so`.

These utilities are thought to be able to send messages to three different destinations: to the standard error, to the Message Logger (using the *congestion-proof* policy) and published in a specific DIM service. To send messages to the Message Logger the parameter `loggerType` must have the first (least significant) bit set.

To the sent messages is pre-pended the header:

```
MMMdd-hhmmss[SEVERITY]hostname: pName: fName():
```

where `pName` is the process name and `fName` is the function name. For example:

```
Aug11-104750[DEBUG]lhcbcn2: tmSrv: startCmndHandler():
```

6.3.1. Synopsis

```
#include “msgUtils.h”

int msgInit(char **argv, int loggerType, char *srvName);
```

6.3.2. Description

Initialize the message stream. Must be called before any `msgSend()` calls.

6.3.3. Arguments

`char **argv`

The argument vector, as passed to the `main(int argc, char **argv)` function.

`int loggerType`

The message destination. Values allowed for **`loggerType`** are in the range 0...7.
The **`loggerType`** value must be the result of a bitwise `OR` of the following values:

0x0	`NOLOG`	Don't write log at all.
0x1	`DIMLOGGER`	Send messages to Message Logger.
0x2	`STDERRLOG`	Send messages to the standard error stream.
0x4	`DIMSVC`	Publish messages to a specific DIM service.

To use `loggerType` 0x4, the `msgInit()` call must be followed by the `msgStart(void)` call or by another `dis_start_serving()` call for the DIM server `/<HOSTNAME>/srvName`.

`char *srvName`

Used only by `loggerType` 0x4. Dummy parameter for the other `loggerType`. Used to set-up the specific DIM service, whose name will be: `/<HOSTNAME>/srvName/log`.

6.3.4. Synopsis

```
#include “msgUtils.h”

int msgSend(char *fName, int severity, char *message);
```

6.3.5. Description

Send a message to the logger destination chosen with `msgInit()`.

6.3.6. Arguments

`char *fName`

The name of the function which send this message.

`int severity`

The severity level of the message. Must be in the range 1…5. The parameters `DEBUG`, `INFO`, `WARN`, `ERROR`, `FATAL`, defined in `msgUtils.h` can be used instead of the number.

`char *message`

The message to be sent.

6.3.7. Example

```
#include "msgUtils.h"
int main(int argc, char** argv)
{
  char message[4000]=”my message”;
  char *fName="main()";
  int loggerType=0x1;
  char srvName[]="";
  msgInit(argv,loggerType,srvName);
  msgSend(fName,INFO,msg);
  return 0;
}
```

The above program sends messages only to the Message Logger.

7. References

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

[2] see fifo(4) manual page and `/usr/src/linux/fs/fifo.c`.

[3] *Circular pipes*, see URL http://lwn.net/Articles/118750/, Linux Kernel Mailing List, *Patch: Make pipe data structure be a circular list of pages, rather than a circular list of one page*, see URL http://lwn.net/Articles/118751/, Linus Torvalds, *Re: Make pipe data structure be a circular list of pages, rather than*, see URL http://lwn.net/Articles/118756/, Linus Torvalds, Re: *Make pipe data structure be a circular list of pages, rather than*, see URL http://lwn.net/Articles/118760/.

[4] F. Bonifazi, D. Bortolotti, A. Carbone, D. Galli, D. Gregori, U. Marconi, G. Peco, V. Vagnoni, *The Task Manager for the LHCb On-Line Farm*, LHCb Technical note 2004-099 DAQ.