Getting started with PM2

The PM2 Team[1]

July 31, 2009 at 15:00

[1]runtime@labri.fr

Abstract

PM2 (*Parallel Multithreaded Machine*) is a distributed multithreaded programming environment designed to efficiently support irregular parallel applications on distributed architectures of SMP nodes. This manual presents an installer's, user's and developer's guide for PM2.

If you have any questions concerning PM2, please, send an email to `pm2-users@lists.gforge.inria.fr`. However, please first make sure that your problem is not already listed in the *Frequently Asked Questions* Section 6.2.

The latest version of this document is available from the following URL:

`http://runtime.bordeaux.inria.fr/pm2-doc/`

Keywords: RPC, Communication, Thread, Distributed Programming.

Contents

Programs

Chapter 1

Introducing PM2

PM2 is developed at LaBRI (*Laboratoire Bordelais de Recherche en Informatique*), a research laboratory located in Bordeaux, France, jointly supported by the INRIA, the CNRS, and the University of Bordeaux 1. Before that, PM2 was developed at LIP (*Laboratoire de l'Informatique du Parallélisme*), a research laboratory located at the ENS Lyon (*Ecole Normale Supérieure de Lyon*), France, jointly supported by the INRIA, the CNRS and the University Claude Bernard Lyon 1. PM2 was originally designed at LIFL, University of Lille, France.

PM2 (*Parallel Multithreaded Machine*) is a distributed multithreaded programming environment designed to support efficiently irregular parallel applications on distributed architectures of SMP nodes. PM2 is primarily designed for medium-size clusters of commodity processing nodes interconnected by high-performance networks. However, nothing in the design of PM2 prevents from using it on massively parallel MIMD machines at one end of the spectrum, or as a support for metacomputing over the Internet on the other end. Actually, a new version of PM2's communication library has been developed to support heterogeneous networking configurations, such as sets of interconnected clusters.

The PM2 software suite is composed of different software components for clusters and clusters of clusters presented in the following sections.

1.1 Runtime Systems

1.1.1 μPM2

μPM2 is a low level generic runtime system which integrates multithreading management, communication management and elementary shared memory management. Its interface is reduced to simple RPC paradigm as well as the classical multithreading primitives. Its implementation is somewhat a software glue in-between the Marcel and Madeleine libraries and a iso-address memory management module. The PM2 environment was rewritten to rely exclusively on this runtime system.

1.2 Communication Libraries

1.2.1 Madeleine

Madeleine is a communication multi-cluster library which implements a concept of communication channel that can be either physical (an abstraction of a physical network) or virtual. In that later case, it is possible to build heterogeneous virtual networks. Madeleine provides a forwarding mechanism relying onto gateways when the configuration allows it (nodes which belong to several physical networks with different technologies). Madeleine is able to dynamically select the best way to transmit data according to the underlying network technology (multi-paradigm). This is made possible by specifying constraints upon data to be sent (concept of programming by contract) and enables to keep a good level of performance on top of technologies with very differing functioning. Madeleine is available on top of various network hardware: Myrinet, SCI, Ethernet or even VIA and runs on the following architectures: Linux/IA32, Linux/Alpha, Linux/Sparc, Linux/PowerPC, Solaris/Sparc, Solaris/IA32, AIX/PowerPC, WindowsNT/IA32.

1.2.2 NewMadeleine

NewMadeleine is a complete redesign and rewrite of Madeleine. The new architecture aims at enabling the use of a much wider range of communication flow optimization techniques. It is entirely modular: The request scheduler itself is interchangeable, allowing experimentations with multiple approaches or on multiple issues with regard to processing communication flows. In particular we implemented an optimizing scheduler called SchedOpt. SchedOpt targets applications with irregular, multi-flow communication schemes such as found in the increasingly common application conglomerates made of multiple programming environments and coupled pieces of code, for instance. SchedOpt itself is easily extensible through the concepts of optimization strategies (what to optimize for, what the optimization goal is) expressed in terms of tactics (how to optimize to reach the optimization goal). Tactics themselves are made of basic communication flows operations such as packet merging or reordering.

1.2.3 Mad-MPI

Mad-MPI is a new light implementation of the MPI standard. This simple, straightforward proof-of-concept implementation is a subset of the MPI API, that allows MPI applications to benefit from the NewMadeleine communication engine.

Mad-MPI also implements some optimizations mechanisms for derived datatypes. MPI derived datatypes deal with noncontiguous memory locations. The advanced optimizations of NewMadeleine allowing to reorder packets lead to a significant gain when sending and receiving data based on derived datatypes.

To have more information about Mad-MPI, please consult the following URL: `http://runtime.bordeaux.inria.fr/MadMPI/`

1.2.4 PadicoTM

PadicoTM is a communication framework for grids, built upon Madeleine. It is able to make a large spectrum of middleware systems (MPI, CORBA, JXTA, SOAP, HLA, JVM, etc.) work

over Madeleine.

To have more information about PadicoTM, please consult the following URL: `http://runtime.bordeaux.inria.fr/PadicoTM/`

1.3 Multithreading Libraries

1.3.1 Marcel

Marcel is a thread library that provides a POSIX-compliant interface and a set of original extensions. It can also be compiled to provide ABI-compabiblity with NTPL threads under Linux, so that multithreaded applications can use Marcel without being recompiled.

Marcel features a two-level thread scheduler (also called N:M scheduler) that achieves the performance of a user-level thread package while being able to exploit multiprocessor machines. The architecture of Marcel was carefully designed to support a high number of threads and to efficiently exploit hierarchical architectures (e.g. multi-core chips, NUMA machines).

So as to avoid the blocking of kernel threads when the application makes blocking system calls, Marcel uses Scheduler Activations when they are available, or just intercepts such blocking calls at dynamic symbols level.

1.3.2 PIOMan

PIOMan is an event detector server that is dedicated to the reactivity, typically communication events. PIOMan provides the NewMadeleine library with methods to detect quickly communication queries.

To have more information about PIOMan, please consult the following URL: `http://runtime.bordeaux.inria.fr/pioman/`

Chapter 2

Installing PM2

This section describes how to install PM2 on your own machine. Please, first check that your system configuration fits the installation pre-requisites, especially with respect to the GNU utilities as mentioned in Section 2.1.3.

2.1 PM2 pre-requisites

PM2 is a highly portable and efficient environment and the current software is yet available on a wide range of architectures. The implementation is built on top of several distinct software components: Marcel and Madeleine. we briefly review the currently supported architectures and systems.

2.1.1 Multithreading: Marcel

Marcel is a POSIX-compliant thread package which provides extra features, such as thread migration. Marcel is currently available on the following platforms, listed in the `ARCHITECTURES` file of the distribution:

```
ravel% more ${PM2_ROOT}/ARCHITECTURES
AIX/RS6K
FREEBSD/X86
IRIX/MIPS
LINUX/ALPHA
LINUX/PPC
LINUX/X86
OSF/ALPHA
SOLARIS/SPARC
SOLARIS/X86
UNICOS/ALPHA
```

2.1.2 Communication: Madeleine

Madeleine is a generic communication interface which is able to fully exploit the low latency and the high bandwidth of high-speed networks such as Myrinet or SCI. This PM2 communication subsystem currently supports the following communication interfaces: TCP, MPI (LAM-MPI, MPI-BIP), VIA, BIP (on top of Myrinet), SISCI (on top of SCI) and SBP (on top of Fast-Ethernet).

2.1.3 Required development tools

As previously mentioned, PM2 has been designed to be easily portable and only relies on the availability of the two following development tools:

- GNU C Compiler `gcc` (version `3.2` and higher). The version `2.95` of `gcc` can also be used, but you need to uncomment the last line in the `pm2/make/config`.
- GNU `make` (version `3.81` and higher).

Also, PM2 includes a large number of shell scripts, which are assumed to be run with the *GNU utilities* such as `expr`, `tr`, `head`, `cut`, `test`. A shell that understands substitutions like `${var/find/replace}`, for instance `bash`, `zsh`, `ksh`, is needed to run these scripts.

2.2 Getting the PM2 software and help

PM2 files are hosted on the InriaGforge server at `https://gforge.inria.fr/projects/pm2/`, also accessible from the PM2 web site at `http://runtime.bordeaux.inria.fr/pm2/`.

Please, send an email to `pm2-users@lists.gforge.inria.fr` if you encounter problems with the PM2 software. However, please first make sure that your problem is not already listed in the *Frequently Asked Questions* Section 6.2.

2.2.1 Getting the latest release

The latest release for the package `pm2` is available from the main page of the project PM2 at `https://gforge.inria.fr/projects/pm2/` under the tab **Files**. It can be extracted with the following command entered at the shell prompt:

```
ravel% tar xvfz pm2.tar.gz          # Using the GNU tar utility
```

2.2.2 Getting the latest version

The source code is managed by a `Subversion` server hosted by the InriaGforge. To get the source code, you need:

1. To install the client side of the software `Subversion` if it is not already available on your system. The software can be obtained from `http://subversion.tigris.org/`.
2. To become a member of the project `pm2`. For this, you first need to get an account to the gForge server. You can then send a request to join the `pm2` project (`https://gforge.inria.fr/project/request.php?group_id=30`).

   You can also choose to check out the project's SVN repository through anonymous access. In that case, you do not need to become a member, and you will have limited access to the repository.

More information on how to get a gForge account, to become a member of a project, or on any other related task can be obtained from the InriaGforge at `https://gforge.inria.fr/`. The most important thing is to upload your public SSH key on the gForge server (see the FAQ at `http://siteadmin.gforge.inria.fr/FAQ.html#Q6` for instructions).

You can now check out the latest version from the `Subversion` server

- using the anonymous access.

```
ravel% svn checkout svn://scm.gforge.inria.fr/svn/pm2/trunk pm2
```

- or using your gForge account.

```
ravel% svn checkout svn+ssh://<login>@scm.gforge.inria.fr/svn/pm2/trunk pm2
```

2.3 The PM2 distribution

Once extracted, the PM2 distribution should be available under the `./pm2/` directory. The PM2 distribution is organized as follows:

`appli` Application configuration

`bin` PM2 scripts

`common` Common configuration scripts to all modules

`doc` Documentation files

`ezflavor` Flavor configuration graphic interface

`generic` Common configuration options

`init`

`leonie` Bootstrap code for PM2 applications

`leoparse` Parser for PM2 configuration files

`mad3` Communication subsystem (obsolete version)

`make` PM2 makefile system

`marcel` Multithreading subsystem

`memory` Memory management system

`modules` Link to all the PM2 modules

`nmad` Communication subsystem (current version)

`ntbx` Generic network management utility toolbox

`pioman` Generic I/O manager

`pm2` PM2 *en personne*!

`profile` PM2 profiling utilities

`puk` Padico micro-kernel

`stackalign` Stack initialization for Marcel

`tbx` Generic PM2 utility toolbox

2.4 Setting the environment variables

Several environment variables have to be set so that PM2 correctly works.

- The `PM2_ROOT` variable contains the path to the PM2 distribution root directory. Setting that variable is **mandatory**. It is the only one needed, all other variables have default values.
- It is highly recommended to add the directory `${PM2_ROOT}/bin` in the active search path.
- The `PM2_HOME` variable contains the path to the PM2 administrative directory. The default is your home directory `${HOME}`.
- By default, configuration files are stored in the `${PM2_HOME}/.pm2` directory. You can change this by setting the `PM2_CONF_DIR` variable to another directory.

  All the files generated within the compilation process will be placed by default in the `${PM2_ROOT}/build` directory, more precisely in the `${PM2_ROOT}/build/${PM2_ASM}`. The environment variable `PM2_ASM` describes the architecture of the machine and is set to the result of the call `pm2-arch --pm2-asm`. If you wish to keep the compiled files into another directory, you can set the `PM2_BUILD_DIR` environment variable to this directory. The files will then be kept in the `${PM2_BUILD_DIR}/${PM2_ASM}` directory.

  Please note that these two directories have to be reachable from any machine used when running PM2 programs. Using the `/tmp` directory would not be suitable!

You may wish to insert the following lines in your `.cshrc` file (if your default shell is `csh`):

```
setenv PM2_HOME          ${HOME}
setenv PM2_ROOT          ${PM2_HOME}/pm2
setenv PATH              ${PATH}:${PM2_ROOT}/bin
```

Warning Please make sure that these variables are correctly set within remote shell commands (i.e., scripts invoked by `ssh`).

2.5 Creating the standard PM2 flavors

You should now create your own, private configuration database. The files are stored in the `${PM2_HOME}/.pm2` directory, and consist of a set of *flavors*. A flavor is simply a (rather complex!) set of options to be passed to the PM2 modules at compilation time and/or at execution time. Flavors are organized so as to let you easily maintain several versions of the PM2 system at the same time: for instance, one version for the TCP protocol and another version for the BIP/Myrinet protocol. Or, one version with the debugging parameters on, and another one with the debugging parameters turned off for performance.

Warning Do not forget to re-source your `.cshrc` file to activate the new environment variables and the new search `PATH`.

```
ravel% source ${HOME}/.cshrc          # If you are running csh
ravel% cd ${PM2_ROOT}
ravel% make init
```

This checks the whole source hierarchy for consistency, and extracts the option sets for each PM2 module if necessary. It then creates the various flavors. You can list them by calling

```
ravel% pm2-flavor list
```

2.6 Selecting your own PM2 flavor

Configuring PM2 is quite straightforward. The underlying platform is automatically detected. Hence, there is no need to specify the operating system/processor pair.

Configuring flavors is done through a set of external utilities to be introduced later. We strongly discourage you to edit the flavor files by hand.

In the following, we will use different flavors of PM2 depending on which aspect of PM2 we wish to use. Selecting a flavor is done by setting the environment variable `PM2_FLAVOR` to the name of the flavor one wishes to use.

```
ravel% setenv PM2_FLAVOR pm2
```

2.7 Specifying the configuration for execution

Before executing a PM2 application, it is needed to specify the list of host names on which the application is going to run. For this, use the `pm2-conf` command. Here I want to use my local machine called `ravel`, and two neighboring ones called `debussy` and `faure`.

```
ravel% setenv PM2_FLAVOR <the flavor you are currently using>
ravel% pm2-conf ravel debussy faure
The current PM2 configuration contains 3 host(s) :
0 : ravel
1 : debussy
2 : faure
```

Note that a configuration is specific to a specific flavor. Using a new flavor will require the setting of a configuration for this flavor.

PM2 will consider that processing node 0 is a process run by `ravel`, node 1 by `debussy` and node 2 by `faure`. Observe that I select my current workstation as node 0. This is for convenience only as PM2 does not require to be executed on the local machine. Indeed, there is no reason why the local machine should get enrolled in the execution of the program. When running a PM2 on a dedicated cluster, the local machine is most preferably outside the configuration.

Warning Just to avoid disturbing problems in the following, please make sure that you local machine is allowed to issue `ssh` commands to each machine in the configuration, and that the directory `${PM2_ROOT}` is actually shared by all of them. You can for example try the following command:

```
ravel% pm2-all 'ls ${PM2_ROOT}'
```

(Note the ' characters!)
This script issues a `ssh` command to each node in the current configuration. In case you do not get the list of the files in your current `${PM2_ROOT}` directory, but get instead nothing or a message similar to `permission denied`, you should then check you can connect to the machines which are in your current configuration.

By default, PM2 is connecting to machines using the `ssh` command. To use another command or to specify parameters for the `ssh` command, you can set the `PM2_RSH` variable.

```
ravel% setenv PM2_RSH "ssh -X -f"
```

If you still cannot get a proper output with the command `pm2-all`, then it could be the remote machines in the configuration are not correctly served by the network file system. PM2 cannot work, as it implicitly assumes that the executable of the program will be found on the remote machines at the same place as it is on the local one.

However if it is not possible to correct such problems, it is still possible to run all three PM2 *virtual nodes* as three processes on your local machine:

```
ravel% pm2-conf ravel ravel ravel
The current PM2 configuration contains 3 host(s) :
0 : ravel
1 : ravel
2 : ravel
```

2.8 Creating the Network configuration file

When loading a PM2 program, a network configuration file is read to determine which network links to use between the different machines involved in the execution. In our case, we suppose our 3 machines are connected using the TCP network. The network configuration file can be created as follows:

```
ravel% cat networks.cfg
networks : ({
    name  : mycluster;
    hosts : (ravel, debussy, faure);
    dev   : tcp;
});
```

You then need to tell PM2 to use your new network configuration file instead of the default one. This is done by setting the `PM2_LEONIE_NETWORK_FILE` variable.

```
ravel% setenv PM2_LEONIE_NETWORK_FILE $PWD/networks.cfg
```

2.9 Compiling your first μPM2 program

Warning Please, make sure that your flavor is actually set to `pm2`:

```
ravel% setenv PM2_FLAVOR pm2
```

The PM2 library should now be fully configured. You are ready to compile and execute the example programs of the distribution. Let's start with the simplest one: *Hello world!*, located in the file `${PM2_ROOT}/pm2/examples/simple/hello.c`.

```
ravel% cd ${PM2_ROOT}/pm2/examples/simple
ravel% make hello
```

The compiled version of the program is automatically placed into your own `build/${PM2_FLAVOR}` directory, but you do need not bother with this detail at this point: PM2 cares about it for you! Use the `pm2-which` command to learn where a file is physically located:

```
ravel% pm2-which hello
```

It remains to actually load and run the program.

```
ravel% pm2-load hello
Hello world!
Hello world!
Hello world!
```

You can see that our program generates several times the same message `Hello world`. Each of the nodes involved in the execution displays once the message with a call to the `printf` Unix routine. The output from all the nodes has been redirected to the console of the starting process.

2.10 Compiling your first Marcel program

Warning Please, make sure that your flavor is actually set to `marcel`:

```
ravel% setenv PM2_FLAVOR marcel
```

You can now compile and execute a sample Marcel application.

```
% cd ${PM2_ROOT}/marcel/examples
% make clean
...
% make sumtime
...
<<< Generating libraries: done
    building sumtime.o
    linking sumtime
% pm2-conf ravel debussy faure
The current PM2 configuration contains 3 host(s) :
0 : ravel
1: debussy
2: faure
% pm2-load sumtime 1000
Sum from 1 to 1000 = 500500
time = 4.829ms
```

2.11 Compiling your first Madeleine program

Warning Please, make sure that your flavor is actually set to `mad3`:

```
ravel% setenv PM2_FLAVOR mad3
```

You can now compile and execute a sample Madeleine application.

```
% cd ${PM2_ROOT}/mad3/examples/
% make clean
% make mad_ping
...
<<< Generating libraries: done
    building mad_ping.o
    linking mad_ping
% pm2-conf ravel debussy
The current PM2 configuration contains 2 host(s) :
0 : ravel
1: debussy
% pm2-load mad_ping
##### ravel
##### debussy
(ravel): My global rank is 0
(debussy): My global rank is 1
The configuration size is = 2
Channel: pm2
The configuration size is = 2
Channel: pm2
My local channel rank is = 0
Channel: pm2
My local channel rank is = 1
ping with = 1
pong with = 0
src|dst|size        |latency     |10^6 B/s|MB/s    |
  0   1            4       10.964    0.365    0.348
...
  0   1      2097152    33431.436   62.730   59.824
Exiting
test series completed
Exiting
```

2.12 Compiling your first NewMadeleine program

Warning Please, make sure that your flavor is actually set to `nmad-mpi`:

```
ravel% setenv PM2_FLAVOR nmad-mpi
```

You can now compile and execute a sample NewMadeleine application.

```
% cd ${PM2_ROOT}/nmad/examples/sched_opt
% make clean
% make sr_ping
...
<<< Generating libraries: done
    building sr_ping.o
    linking sr_ping
% pm2-conf ravel debussy faure
The current PM2 configuration contains 3 host(s) :
0 : ravel
1: debussy
2: faure
% pm2-load sr_ping
##### ravel
##### debussy
##### faure
[2] receiving from node 1, gate 1
[1] receiving from node 0, gate 0
[0] sending to node 1, gate 0
[0] receiving from node 2, gate 1
[1] sending to node 2, gate 1
[2] sending to node 0, gate 0
[0] Message [12.450000,3.140000]
%
```

2.13 Solving problems

If something goes wrong at any point, you can always activate the emergency repair tool provided by PM2.

```
ravel% cd ${PM2_ROOT}
ravel% make sos
```

This results into a listing of your current configuration, attempts to rebuild the flavor database and clears the compiled PM2 library as cleanly as possible. It should then restore a fresh and safe configuration for you, ready to restart the whole compilation. The output of the command should look like:

```
********* Checking environment variables *********
PM2_HOME =
FLAVOR = pm2
CURDIR = .../pm2
PM2_ROOT = .../pm2
PM2_BUILD_DIR = .../pm2/build
********* Refreshing files for current flavor *********
make[1]: Entering directory '.../pm2'
Re-generating flavor pm2...
flavor 'pm2' unmodified
Cleaning for flavor pm2...
make[1]: Leaving directory '.../pm2'
Humm... Well, all should be ok now!
```

Note that the final message is automatically generated!

Chapter 3

Discovering PM2

This chapter aims at illustrating the basic features of PM2 through some very simple programs. The next chapters present more advanced features that may be needed when developing *real* parallel applications.

3.1 Compiling your own PM2 program

You may compile you own PM2 program just as a usual C program. You just need to make sure the necessary definitions and libraries are included. As the command line is rather complex, PM2 provides a simple utility to generate it online:

- `pm2-config --cc` generates the name of the adequate C compiler (most probably, `gcc`),
- `pm2-config --cflags` generates the necessary `CFLAGS`,
- and `pm2-config --libs` generates the list of libraries to be searched on linking.

Thus, the standard command line to compile a PM2 program `hello.c` looks like:

```
# Assuming you use csh...
setenv CC       "`pm2-config --cc`"
setenv CFLAGS   "`pm2-config --cflags`"
setenv LIBS     "`pm2-config --libs`"
$CC $CFLAGS hello.c $LIBS -o hello
```

Observe that the source file `hello.c` should be mentioned *before* the PM2 libraries, as the external symbols are searched by `gcc` in the libraries from left to right.

Well... Such a command line is rather tedious to enter! You will find on Program 3.1 a `Makefile` which does all the work for you. Executing `make hello` will compile the source file `hello.c` with all the necessary parameters.

Observe that using dynamic calls to the `pm2-config` utility guarantees that you are compiling with the suitable options with respect to the current value of the flavor, as specified by the `PM2_FLAVOR` shell variable.

Let us start our PM2 tour by writing the traditional "*Hello World!*" program. Then, we will extend it step by step to cover the main functionalities provided by the different PM2 programming interfaces. We will in the following sections present the Hello World! program written on top of Madeleine, Marcel, and μPM2.

Program 3.1 A minimal GNU Makefile for PM2 programs

```
## This Makefile should be run with GNU make

ifndef PM2_FLAVOR
$(error PM2_FLAVOR is not yet defined!)
endif

CFLAGS          := $(shell pm2-config --cflags)
LDFLAGS         := $(shell pm2-config --libs)
CC              := $(shell pm2-config --cc)

.PHONY: all clean

CFILES          := $(wildcard *.c)
OFILES          := $(CFILES:%.c=%.o)
EXEFILES        := $(CFILES:%.c=%)

ifdef PM2_BUILD_DIR
TARGET  :=      ${PM2_BUILD_DIR}/${PM2_ASM}/${PM2_FLAVOR}/examples/bin
else
TARGET  :=      ${PM2_ROOT}/build/${PM2_ASM}/${PM2_FLAVOR}/examples/bin
endif

all: ${EXEFILES}

${TARGET}:
        -@mkdir ${TARGET}

${EXEFILES}: %: %.c ${TARGET}
        @echo "Compiling $< for flavor '${PM2_FLAVOR}'..."
        make -C ${PM2_ROOT}
        @${CC} ${CFLAGS} $< ${LDFLAGS} -o ${TARGET}/$@

clean:
        -rm ${EXEFILES} ${OFILES}
```

3.2 Initializing and terminating a PM2 program

Initialization and termination phases are similar to all PM2 programs, whatever the programming interface you wish to use: Marcel, Madeleine or μPM2. This section will explain these two phases.

3.2.1 The header files

A PM2 program must always include the PM2-specific `pm2_common.h` header file, along with other standard header files. Note that this is the *only* PM2 header file that has to be included by user applications.

3.2.2 Initialization

A PM2 program has to call the two functions `common_pre_init` and `common_post_init` to effectively initialize the PM2 runtime system. Moreover, this step involves a global syn-

chronization phase (actually, a global synchronization barrier) among all the PM2 nodes so that each process is assigned a unique *rank number* on return from the functions. In consequence, it makes no sense to call functions concerned with the global execution environment (such as node numbers, etc.) *before* this point. Please, refer to Section 3.5.3 for details.

The first two arguments of the function are the usual pair `argc/argv` of the C `main` function. The type of the last argument is `common_attr_t`, it is used to specify parameters for the configuration of the application, in most cases, the value `NULL` will do.

Most importantly, the initialization functions spawn a number of internal *thread daemons* that are in charge of listening to the network and answering to external requests such as RPCs, incoming thread migrations, etc. A node should be ready to handle all possible incoming requests from any other node at this point. As a consequence, it is not safe to do any initialization *after* this point, as the user has no control about the interleaving of requests and the relative speed of the nodes. Even though the initialization call would be placed just after the initialization functions call, an arbitrary delay may occur between the two successive instructions! It follows that if some initialization code needs to be performed before any thread is started, then this code *must* be called *before* calling the initialization functions.

Also note that the Unix standard input/output streams may not be correctly initialized before the call to the initialization functions. Thus, the behavior of programs using I/O operations before the initialization is not defined.

3.2.3 Termination

To exit from a PM2 session, each node must call the `common_exit` routine. The parameter is similar to the one for the `common_pre_init` function.

Note that the termination of any PM2 program is a tricky task as PM2 does not perform any automatic termination detection. Thus, the termination decision must be made at the application level.

3.3 A minimal Madeleine program: Hello World!

Program 3.2 shows a minimal Madeleine program. Compile it:

```
ravel% setenv PM2_FLAVOR mad3
ravel% cd $PM2_ROOT/mad3/examples
ravel% make gs_mad_hello
[...]   # Normally, no warning whatsoever!
```

You can also choose to do the compilation by using the Makefile given in Program 3.1. It will create the executable file in the appropriate PM2 build directory, e.g `${PM2_BUILD_DIR}/${PM2_ASM}/`. When needed, the program will be located using the utility `pm2-which`.

```
ravel% pm2-which gs_mad_hello
/home/nfurmento/build/i386/mad3/examples/bin/gs_mad_hello
```

You can now execute the application:

```
ravel% pm2-conf ravel debussy
The current PM2 configuration contains 2 host(s) :
0 : ravel
1 : debussy

ravel% pm2-load gs_mad_hello
##### ravel
##### debussy
success
```

3.3.1 Informations about the application

In order to have a global view of the application, each node needs to retrieve the Madeleine object which contains for example informations about the communication channel, the set of processes in this channel. These informations will allow each node to retrieve its local rank in the current channel. Here a set of instructions showing how to get these informations:

```
/* The communication channel */
p_mad_channel_t channel = NULL;
/* The 'local' rank of the process in the current channel */
ntbx_process_lrank_t my_local_rank = -1;
/* The set of processes in the current channel */
p_ntbx_process_container_t pc = NULL;
/* The globally unique process rank  */
ntbx_process_grank_t process_rank;

/* Retrieve the Madeleine object */
p_mad_madeleine_t madeleine = mad_get_madeleine();

/* Get a reference to the corresponding channel structure */
channel = tbx_htable_get(madeleine->channel_htable, CHANNEL_NAME);
/* If that fails, it means that our process does not */
/* belong to the channel */
if (!channel) {
  DISP("I don't belong to this channel");
  return;
}

/* Get the set of processes in the channel */
pc = channel->pc;

/* Convert the globally unique process rank to its */
/* _local_ rank in the current channel */
process_rank  = madeleine->session->process_rank;
my_local_rank = ntbx_pc_global_to_local(pc, process_rank);
```

3.3.2 Packing and unpacking messages

The packing model of PM2 has been carefully designed so as to enable high-performance communication on modern Gigabit network interfaces such as BIP/Myrinet, SISCI/SCI, VIA, etc. In this context, it is of uttermost significance to avoid copies: actually, the time for copying a buffer within a node is of the same order of magnitude as the time for sending it over the network to some remote node! The key for performance is therefore to allow for *zero-copy* communication: the message has to be directly taken from its initial location

in user-space, and directly placed into its final destination in user-space, without any additional copy. Designing communication interfaces which can efficiently deal with messages featuring a complex structure, or messages of unpredictable size, is a difficult task. The approach of PM2, or more accurately of its underlying communication library Madeleine, is to control the packing and unpacking operations with additional *flags*.

A Madeleine message consists of several pieces of data, located anywhere in user-space. It is constructed (resp. de-constructed) incrementally using *packing* (resp. *unpacking*) routines, possibly at multiple software levels without losing efficiency. The following example illustrates the power of the Madeleine interface. Let us consider a remote procedure call which takes an array of unpredictable size as a parameter. When the request reaches the destination node, the header is examined both by the multithreaded runtime (to allocate the appropriate thread stack and then to spawn the server thread) and by the user application (to allocate the memory where the array should be stored).

The critical point of a sending operation is obviously the series of *packing* calls. Such packing operations simply *virtually* append the piece of data to a message under construction. In addition to the address of data and its size, the packing primitive features a *flag* parameter which specifies the behaviour of the operation. Available sending flags are defined as follows:

`mad_send_SAFER` This flag indicates that PM2 should pack the data in a way that further modifications to the corresponding memory area should not corrupt the message. This is particularly mandatory if the data location is reused before the message is actually sent.

`mad_send_LATER` This flag indicates that PM2 should not consider accessing the value of the corresponding data until the `mad_end_packing` primitive is called. This means that any modification of these data between their packing and their sending shall actually update the message contents.

`mad_send_CHEAPER` This is the default flag. It allows PM2 to do its best to handle the data as efficiently as possible. The counterpart is that no assumption should be made about the way PM2 will access the data. Thus, the corresponding data should be left unchanged until the send operation has completed. Note that most data transmissions involved in parallel applications can accommodate this `mad_send_CHEAPER` behaviour.

The following flags control the reception of user data packets:

`mad_receive_EXPRESS` This flag forces PM2 to guarantee that the corresponding data are immediately available after the the *unpacking* operation. Typically, this flag is mandatory when the data is needed to determine the next forthcoming *unpacking* calls. On some network protocols, this functionality may be available for free. On some others, it could penalize the latency and the bandwidth. Users should therefore extract data using this behaviour only when necessary.

`mad_receive_CHEAPER` This flag allows PM2 to defer the extraction of the corresponding data until the execution of the `mad_end_unpacking` routine. Thus, no assumption can be made about the exact moment at which the data will be extracted. Depending on the underlying network protocol, PM2 will do its best to minimize the overall message transmission time. If combined with `mad_send_CHEAPER`, this flag always guarantees that the corresponding data is transmitted as efficiently as possible.

Observe that the emission and reception flags should be *both* specified by the matching packing/unpacking calls, and these specifications should be identical. Again, unspecified behavior would result from mismatching flags.

In Program 3.3, everything is sent `CHEAPER`, so as to save time. The user must be careful not to corrupt the variables `len`, `s` and `s2` before calling `mad_end_packing`. On the reception side, the variable `len` *must* be received `EXPRESS`, as its value is needed to allocate the buffer for the unpacking of the variables `s` and `s2`. Then, these two variables can be received `CHEAPER`. If the underlying operating system and the network interface permit, then this (possibly large) message will thus be directly installed into the reception buffer without any extra copy, providing the user with optimal performances.

3.3.3 Sending and receiving messages

In order to sent a message to another node, you need to know the channel it belongs to, and its local rank within this channel. You can then open an outgoing connection with this node by calling `mad_begin_packing`, pack data in the message as explained in the previous section, and finalize the message by calling `mad_end_packing`.

Nodes can receive messages sent on a specific channel by calling `mad_begin_unpacking`. Note here that is not possible to specify which node you want to receive a message from. If two nodes belonging to the same channel send a message to a third node, then receiving a message on this third node will either read the message from the first or the second node. After calling `mad_begin_unpacking`, data have to be unpacked as explained in the previous section, a call to `mad_end_unpacking` will finalize the reception of the message.

Program 3.2 Minimal Madeleine program

```
/*
 * PM2: Parallel Multithreaded Machine
 * Copyright (C) 2001 "the PM2 team" (see AUTHORS file)
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or (at
 * your option) any later version.
 *
 * This program is distributed in the hope that it will be useful, but
 * WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
 * General Public License for more details.
 */

#include "pm2_common.h"

void do_main_code(p_mad_madeleine_t madeleine);

int main(int argc, char *argv[]) {
  common_pre_init(&argc, argv, NULL);
  common_post_init(&argc, argv, NULL);

  do_main_code(mad_get_madeleine());
  common_exit(NULL);
  exit(EXIT_SUCCESS);
}

void do_main_code(p_mad_madeleine_t madeleine) {
  p_mad_channel_t channel = NULL;
  ntbx_process_lrank_t my_local_rank = -1;
  p_mad_connection_t out = NULL;
  p_mad_connection_t in = NULL;

  channel = tbx_htable_get(madeleine->channel_htable, "pm2");
  my_local_rank = ntbx_pc_global_to_local(channel->pc,
                                          madeleine->session->process_rank);

  if (my_local_rank == 0) {
    char hostname[100];
    char data[50] = "";
    char *rdata = NULL;
    int rlen, len = 0;

    strcpy(data, "Hello World from ");   /* Prepare the data to be sent */
    gethostname(hostname, 100);
    strcat(data, hostname);
    len = strlen(data);

    out = mad_begin_packing(channel, 1); /* Open the out connection */
    mad_pack(out, &len, sizeof(len), mad_send_SAFER, mad_receive_EXPRESS);
    mad_pack(out, data, len, mad_send_CHEAPER, mad_receive_CHEAPER);
    mad_end_packing(out);                /* End of the sent */

    in = mad_begin_unpacking(channel);   /* Starts the reception */
    mad_unpack(in, &rlen, sizeof(rlen), mad_send_SAFER, mad_receive_EXPRESS);
    rdata = TBX_CALLOC(1, rlen);
    mad_unpack(in, rdata, rlen, mad_send_CHEAPER, mad_receive_CHEAPER);
    mad_end_unpacking(in);               /* Ends the reception. */

    if (strcmp(data, rdata) == 0)
      printf("success\n");
    else
      printf("failure. Strings differ: <%s> != <%s>\n", data, rdata);
  }
```

Program 3.3 Sending and receiving data with Madeleine

```
1  void sendData() {
     char s[]  = "A la recherche du temps perdu.";
     char s2[] = "Les carottes sont tres cuites.";
     int len;
5
     len = strlen(s) + 1;

     mad_begin_packing(...);
     mad_pack(..., &len, sizeof(int), mad_send_CHEAPER, mad_receive_EXPRESS);
10   mad_pack(..., s, len, mad_send_CHEAPER, mad_receive_CHEAPER);
     mad_pack(..., s2, len, mad_send_CHEAPER, mad_receive_CHEAPER);
     mad_end_packing(...);
   }

15 void receiveData() {
     int len;
     char *s;
     char *s2;

20   mad_begin_unpacking(...);
     mad_unpack(..., &len, sizeof(int), mad_send_CHEAPER, mad_receive_EXPRESS);
     s = malloc(len);
     mad_unpack(..., s, len, mad_send_CHEAPER, mad_receive_CHEAPER);
     s2 = malloc(len);
25   mad_unpack(..., s2, len, mad_send_CHEAPER, mad_receive_CHEAPER);
     mad_end_unpacking(...);
   }
```

3.4 A minimal Marcel program: Sum of the first n integers

Program 3.4 shows a minimal Marcel program. Compile it and run it:

```
ravel% setenv PM2_FLAVOR marcel
ravel% cd $PM2_ROOT/marcel/examples
ravel% make gs_sumtime
[...]   # Normally, no warning whatsoever!
ravel% pm2-load gs_sumtime
Sum from 1 to 1000 = 500500
```

3.4.1 The marcel_main function

The main function of a Marcel program is named `marcel_main`, in contrast to traditional C programs which use the well-known `main` function name. In fact, the *real* `main` function of the program is provided by the PM2 libraries. It has to set up the execution environment before calling the user `marcel_main` function. This allows PM2 to greatly enhance the performance of various thread management functions. The arguments remain the regular `argc/argv` pair, with their usual meaning.

Although most programs (or rather, programmers!) can accommodate such a violation of the usual *C convention*, there exists some applications that require to use the regular `main` function name. In particular, it may be the case with applications linked with non-C code (*e.g.*, Fortran code). In this case, you must use the `stackalign` module, please refer to Section **??** for details.

3.4.2 Thread management

Note TODO...

3.4.3 Semaphores and mutexes

Note TODO...

Program 3.4 Minimal Marcel program

```
/*
 * PM2: Parallel Multithreaded Machine
 * Copyright (C) 2001 "the PM2 team" (see AUTHORS file)
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or (at
 * your option) any later version.
 *
 * This program is distributed in the hope that it will be useful, but
 * WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
 * General Public License for more details.
 */

#include <pm2_common.h>
#include <stdio.h>
#include <stdlib.h>

typedef struct {
  int inf, sup, res;
  marcel_sem_t sem;
} job;

static void job_init(job *j, int inf, int sup) {
  j->inf = inf;
  j->sup = sup;
  marcel_sem_init(&j->sem, 0);
}

static any_t sum(any_t arg) {
  job *j = (job *)arg;
  job j1, j2;

  if(j->inf == j->sup) {
    j->res = j->inf;
    marcel_sem_V(&j->sem);
    return NULL;
  }

  job_init(&j1, j->inf, (j->inf+j->sup)/2 );
  job_init(&j2, (j->inf+j->sup)/2+1, j->sup);

  marcel_create(NULL, NULL, sum, (any_t)&j1);
  marcel_create(NULL, NULL, sum, (any_t)&j2);
  marcel_sem_P(&j1.sem);
  marcel_sem_P(&j2.sem);

  j->res = j1.res+j2.res;
  marcel_sem_V(&j->sem);
  return NULL;
}

int marcel_main(int argc, char **argv) {
  job j;

  common_pre_init(&argc, argv, NULL);
  common_post_init(&argc, argv, NULL);
```



```
  marcel_sem_init(&j.sem, 0);
  j.inf = 1;
  j.sup = 1000;

  marcel_create(NULL, NULL, sum, (any_t)&j);
  marcel_sem_P(&j.sem);
```

3.5 A minimal μPM2 program: Hello World!

Program 3.5 shows an example of a minimal μPM2 code. Compile it and run it:

```
ravel% setenv PM2_FLAVOR pm2
ravel% make hello
[...]   # Normally, no warning whatsoever!
ravel% pm2-conf ravel debussy faure
The current PM2 configuration contains 3 host(s) :
0 : ravel
1 : debussy
2 : faure

ravel% pm2-load hello
Hello World!
Hello World!
Hello World!
```

Program 3.5 Minimal μPM2 program

```
#include "pm2_common.h"

int pm2_main(int argc, char *argv[]) {
  common_pre_init(&argc, argv, NULL);
  common_post_init(&argc, argv, NULL);

  tprintf("Hello World!\n");

  if (pm2_self() == 0)
    pm2_halt();

  common_exit(NULL);
  exit(EXIT_SUCCESS);
}
```

3.5.1 The pm2_main function

The main function of the μPM2 program is named `pm2_main`, in contrast to traditional C programs which use the well-known `main` function name. In fact, the *real* `main` function of the program is provided by the PM2 libraries. It has to set up the execution environment before calling the user `pm2_main` function. This allows PM2 to greatly enhance the performance of various thread management functions. The arguments remain the regular `argc/argv` pair, with their usual meaning.

Although most programs (or rather, programmers!) can accommodate such a violation of the usual *C convention*, there exists some applications that require to use the regular `main` function name. In particular, it may be the case with applications linked with non-C code (*e.g.*, Fortran code). In this case, please refer to Section **??** for details.

3.5.2 Termination

In the case of μPM2, it is important to have a synchronization phase before stopping the nodes. This synchronization phase is achieved by calling the `pm2_halt` routine, it should be

called by exactly one node of the application. It performs a broadcast that requires all nodes to stop answering to requests from the outside world. In fact, this step cuts the links between nodes: it stops the internal daemon threads that are in charge of polling the network. Note that after a node receives this order, it still continues its normal execution in a *standalone* mode: *halting* is no *killing*!

In a second step, all the nodes can then call the usual `common_exit` routine. That will block the calling thread until all other threads (belonging to the same node) terminate. In our example, no other application thread except from the main one is running. Thus, this second step completes as soon as the internal daemon threads are stopped. Without the first step being performed, all nodes would remain hanging, waiting for ever for external requests!

As the call to `common_exit` is potentially blocking, the thread in charge of calling `pm2_halt` should do it first. However, observe that it may well be the case that at a node, some thread is in charge of calling `pm2_halt` and another one `common_exit`, so that the order may in fact be irrelevant.

From the user point of view, the μPM2 program terminates as soon as the shell prompts for the next command from the terminal. This corresponds to the end of the main node. However, note that some nodes may actually keep running after the main node has completed.

3.5.3 Who's who?

Each processing node (that is, Unix process) taking part in a given execution receives its own unique *rank* number. This is an `unsigned int` between 0 and `pm2_config_size() - 1`. A node can get its own rank by calling the `pm2_self` routine.

The processing node with rank 0 has a particular status because it is the only one which whose input/output streams are directly linked to the terminal the application was launched from. We later refer to this process as the *main node* of an application. As a consequence, only the main node of an application can access its standard input stream (*e.g.*, using `scanf`, `gets`, etc.)

As an example on how to use `pm2_self`, the program performs a test on the rank of the current process: if the current process is the main process, then the `pm2_halt` routine is called.

3.6 A minimal μPM2 program: Hello World! using Remote Procedure Calls

The goal of this section is to let you issue a *Remote Procedure Call* in μPM2. We start with a basic scheme, and we refine it step by step so as to demonstrate the versatility of μPM2.

3.6.1 Invoking a remote service

A *service* is a function located on a *server* node, which can be invoked by some *client* node. Observe that the client node may be the same as the server node. It may also be run on the same processor, or on a sibling processor on the same SMP board, or on some remote one. All of this is fully transparent for PM2 users. Nodes are specified through their num-

bers, as allocated by the `pm2-conf` command. In this part, we consider our usual 3-node configuration:

```
ravel% pm2-conf ravel debussy faure
The current PM2 configuration contains 3 host(s) :
0 : ravel
1 : debussy
2 : faure
```

Program 3.6 Defining a service in PM2

```
#include "pm2_common.h"

static int service_id;

static void service(void) {
  pm2_rawrpc_waitdata();
  tprintf("Hello, World!\n");
}

int pm2_main(int argc, char *argv[]) {
  pm2_rawrpc_register(&service_id, service);

  common_pre_init(&argc, argv, NULL);
  common_post_init(&argc, argv, NULL);

  if (pm2_self() == 0) {
    pm2_rawrpc_begin(1, service_id, NULL);
    pm2_rawrpc_end();

    pm2_halt();
  }

  common_exit(NULL);
  exit(EXIT_SUCCESS);
}
```

In Program 3.6, a basic service is defined. Its behavior is to print `Hello, World!` on the standard output of the node which is executing it. At the level of PM2, a *service* is just an `int` associated to some function using the `pm2_rawrpc_register` routine. Observe that the service function takes no argument.

Of course, all the available servers have to be registered *before* they may be invoked. The association between the service identifier and the service function has thus to be set up *before* calling the PM2 initialization routines. Doing otherwise will result in unspecified behavior, probably depending on the relative speed of the nodes. The user is therefore *strongly advised* to give extra attention to this requirement.

Returning to the program, the processing node 0 invokes the remote service `service_id` on the processing node 1. The service invocation is initiated using the `pm2_rawrpc_begin` routine. The first argument of the routine is the unique number of the server node. The second argument is the service identifier. It has to be a valid number on the *server node*. The third argument is used for specifying additional *attributes* to be discussed later. In this basic case, no additional data is to be provided together with the service invocation, and `NULL` is a sensible default. The `pm2_rawrpc_end` routine finalizes the service invocation. Returning

from this routine guarantees that the service invocation has been completed from the client's point of view.

In this example, the service is synchronous: the `pm2_rawrpc_end` returns in the client node only *after* the service function has returned on the remote server node. Thus, the client node can safely call `pm2_halt` to force the termination of the distributed program.

It is now time to run the program:

```
ravel% pm2-load rpc
Hello, World!
```

The careful reader will have noticed that the definition of the service function includes a call to the `pm2_rawrpc_waitdata` routine. This is necessary to instruct the service function that no additional data is to be expected, so that it can proceed safely. Actually, the service function has no way of discovering that the RPC has been invoked without any additional data, and it is up to the user to build the program client and server parts of the program in a consistent way. More about this subject in the next section!

3.6.2 Passing parameters

Let us see now the case of a service function invoked with additional parameters. In Program 3.7, node 0 invokes a service on node 1 with a string as a parameter. The service consists namely in printing the string on the standard output.

Which string? Well, the original motivation of PM2 was to provide a runtime environment for high-performance distributed programs with a highly irregular behavior: branch-and-bound search, computation on sparse matrices, etc. In one word: In Search of Lost Time, *la recherche du temps perdu!* You may remember that a *Madeleine*, a typically French (delicious) cookie, played a central role in the life of *Marcel* Proust...

On the client side, the parameters of the service are *packed* together between the `pm2_rawrpc_begin` and the `pm2_rawrpc_end` calls, using the `pm2_pack_*` routine family. You can pack integers, byte arrays, etc. (even pointers!) very much as in PVM or PMI. On the server side, they are *unpacked* within the service function. Again, the service function has no way of guessing what kind of parameters have been packed by the client, nor how many of them have been packed. In short, PM2 messages are *not self-described*. This choice is motivated by performance considerations: including the description of the objects together with the objects into the messages yields a significant overhead for small messages. In this respect, PM2 follows the choice of many other communication interfaces. The counterpart is that the user is responsible for the consistency of the series of packing and unpacking actions. No verification is made at the level of PM2, and unspecified behavior will result in any kind of inconsistency in types and numbers. The user specifies the end of reception on the server side by a call to the `pm2_rawrpc_waitdata` routine.

Let us now run the program.

```
ravel% pm2-load rpc-params
The sentence is: A la recherche du temps perdu.
```

Warning Program 3.7 and similar programs may not work properly when used with communication interfaces which require some specific alignment for communication buffers: for instance, BIP/Myrinet and SISCI/SCI. In this

Program 3.7 Service with a string parameter

```
#include "pm2_common.h"

static int service_id;

static void service(void) {
  int len;
  char *s;

  pm2_unpack_int(SEND_CHEAPER, RECV_EXPRESS, &len, 1);
  s = malloc(len);
  pm2_unpack_byte(SEND_CHEAPER, RECV_CHEAPER, s, len);
  pm2_rawrpc_waitdata();

  tprintf("The sentence is: %s\n", s);
}

int pm2_main(int argc, char *argv[]) {
  pm2_rawrpc_register(&service_id, service);
  common_pre_init(&argc, argv, NULL);
  common_post_init(&argc, argv, NULL);

  if (pm2_self() == 0) {
    /* Warning: this may not work on BIP and SCI */
    char s[] = "A la recherche du temps perdu.";
    int len;

    len = strlen(s) + 1;
    pm2_rawrpc_begin(1, service_id, NULL);
    pm2_pack_int(SEND_CHEAPER, RECV_EXPRESS, &len, 1);
    pm2_pack_byte(SEND_CHEAPER, RECV_CHEAPER, s, len);
    pm2_rawrpc_end();

    pm2_halt();
  }

  common_exit(NULL);
  exit(EXIT_SUCCESS);
}
```

case, variable `s` should be aligned to a proper boundary. This can be done for instance as follows using the specific attributes of `gcc`:

```
#define __ALIGNED__ __attribute__ ((aligned (sizeof(int))))
char s[16] __ALIGNED__;
```

3.6.3 Packing and unpacking data

The packing/unpacking API of μPM2 is identical to the Madeleine API presented in Section 3.3.2. The sending and receiving flags for μPM2 are as follows: SEND_SAFER, SEND_LATER, SEND_CHEAPER, RECV_EXPRESS, RECV_CHEAPER.

On the sender side, `pm2_rawrpc_end` is used to signal the end of the data to be packed, and on the receiver side, `pm2_rawrpc_waitdata` signals the end of the reception of the

data.

Note that the PM2 primitives to send data to a remote service are written on top of the Madeleine primitives.

3.6.4 Threaded services

Service handlers are executed sequentially. As long as a service function has not returned on the server, the service is not available for another client. This may result into deadlocks if the service includes some potentially blocking actions, such as invoking other services, either explicitly or implicitly.

Program 3.8 Spawning a fresh thread to serve a remote request

```
#include "pm2_common.h"

static int service_id;

#define SIZE 1024
char msg[SIZE];

static void f(void *arg) {
  pm2_unpack_byte(SEND_CHEAPER, RECV_CHEAPER, msg, SIZE);
  pm2_rawrpc_waitdata();

  tprintf("%s\n", msg);

  pm2_halt();
}

static void service(void) {
  pm2_service_thread_create(f, NULL);
}

int pm2_main(int argc, char *argv[]) {
  pm2_rawrpc_register(&service_id, service);
  common_pre_init(&argc, argv, NULL);
  common_post_init(&argc, argv, NULL);

  if (pm2_self() == 0) {
    strcpy(msg, "Hello world!");

    pm2_rawrpc_begin(1, service_id, NULL);
    pm2_pack_byte(SEND_CHEAPER, RECV_CHEAPER, msg, SIZE);
    pm2_rawrpc_end();

    /* pm2_halt(); Incorrect!!! */
  }

  common_exit(NULL);
  exit(EXIT_SUCCESS);
}
```

Such problems can be addressed by executing services into a fresh thread instead of the original service handler. This technique is demonstrated in Program 3.8. The service function `service` spawns a fresh thread each time it is invoked through a RPC request.

Then, it immediately returns. The child thread is in charge of extracting the message from the network through a series of `unpack` calls in the usual way.

Observe that PM2 does not care about *who* extracts a message at a node. Any thread can do it, as soon as it has been created with a special creation routine `pm2_service_thread_create`, instead of the regular `pm2_thread_create` one. The only requirement enforced within the runtime system is that the same thread unpacks the series of data pieces and calls the `pm2_rawrpc_waitdata` routine. Having two or more threads extracting data from the network concurrently would actually not make sense: it may raise a run-time error.

Note Luc to Olivier + Raymond: Is it right? A sentence should probably be inserted here to warn that this behavior may be revised in the future versions.

This feature enables the newly spawn thread to do the work on behalf of the original service handler. Yet, this raises a new problem. Actually, the service handler immediately returns after spawning the thread, and then looses any control on its progression. Thus, the `pm2_rawrpc_end` routine on the client side may return *before* the service thread has even started any unpacking whatsoever!

This may result into an incorrect behavior, as demonstrated by the following scenario. Consider that the client is very fast. After exiting the `pm2_rawrpc_end` routine, it calls the `pm2_halt` routine. This triggers the broadcasting of a termination request to all the nodes. Assume now that the messages used for this broadcast do not travel on the same *channel* as the regular messages, so that they can take over and arrive *before* the service thread initiates the unpacking. Then, all the reception facilities of the server node are closed down. When the daemon service thread finally calls the `pm2_unpack_*` routine, nobody remains living here to physically extract the data from the network, and a run-time error results!

Well, you may argue that this scenario is just like SF and little green men attacking the White House... The writer's personal experience is that the worst case is quite common in this matter, in opposite (?) to other domain of life. Just don't try it!

How can one circumvent the problem? A generic technique called *completion* will be introduced later. In the specific example shown in Program 3.8, a simple way-out is to let the service thread issue the call to the `pm2_halt` routine, as it is guaranteed that this thread issues the inter-thread last interaction.

Let us now run the program.

```
ravel% pm2-load rpc-threads
Hello world!
```

To have a bit more fun on closing this part, let us design a slightly more complex program, as the one on Program 3.9. Node 0 issues a RPC request to node 1. On node 1, servicing this request consists in spawning a fresh thread and re-issuing the request to node 2 after having signed it with its name. Similarly for node 2 with respect to node 0. Finally, node 0 serves the request by printing the string to the standard output and triggers termination.

Executing this program should produce a output as follow

```
Passing on string: Init 1 2
Passing on string: Init 1
Sending string: Init
Received back string: Init 1 2
```

Now is a good point to clarify the notion of *node* used throughout this presentation. Actually, the nodes we are considering here are *virtual*: they only refer to Unix processes

located on physical nodes. It looks like common sense practice to use exactly one virtual node per physical node, so that the two notions just match together. However nothing in PM2 requires such practice. PM2 virtual nodes may be located at any physical nodes. The association between PM2 virtual and physical node is done using the `pm2-conf` command. For instance, saying

```
ravel% pm2-conf ravel debussy faure ravel debussy faure
The current PM2 configuration contains 6 host(s) :
0 : ravel
1 : debussy
2 : faure
3 : ravel
4 : debussy
5 : faure
```

is fully correct. Two PM2 processes will be launched on each of the three machines. From the point of view of PM2 programs, this makes no difference.

You may even start all the processes on the same machine!

```
ravel% pm2-conf ravel ravel ravel ravel ravel ravel
The current PM2 configuration contains 6 host(s) :
0 : ravel
1 : ravel
2 : ravel
3 : ravel
4 : ravel
5 : ravel

ravel% pm2-load rpc-threads1
Passing on string: Init 1
Passing on string: Init 1 2
Passing on string: Init 1 2 3 4 5
Passing on string: Init 1 2 3
Sending string: Init
Received back string: Init 1 2 3 4 5
Passing on string: Init 1 2 3 4
```

Also, the running machine does not have to be the one you are logged on. Any machine you have access to can do it as well!

```
ravel% pm2-conf debussy debussy debussy debussy debussy debussy
The current PM2 configuration contains 6 host(s) :
0 : debussy
1 : debussy
2 : debussy
3 : debussy
4 : debussy
5 : debussy

ravel% pm2-load rpc-threads1
Passing on string: Init 1
Passing on string: Init 1 2
Passing on string: Init 1 2 3 4 5
Passing on string: Init 1 2 3
Sending string: Init
Received back string: Init 1 2 3 4 5
Passing on string: Init 1 2 3 4
```

Program 3.9 Spawning threads to serve requests, an extended example

```
#include "pm2_common.h"

static int service_id;

#define SIZE 1024

static void f(void *arg) {
  char rec_msg[SIZE];
  char sent_msg[SIZE];

  int next = (pm2_self() + 1) % pm2_config_size();

  pm2_unpack_byte(SEND_CHEAPER, RECV_CHEAPER, rec_msg, SIZE);
  pm2_rawrpc_waitdata();

  if (pm2_self() == 0) {
    tprintf("Received back string: %s\n", rec_msg);
    pm2_halt();
  }
  else {
    sprintf(sent_msg, "%s %d", rec_msg, pm2_self());
    tprintf("Passing on string: %s\n", sent_msg);

    pm2_rawrpc_begin(next, service_id, NULL);
    pm2_pack_byte(SEND_CHEAPER, RECV_CHEAPER, sent_msg, SIZE);
    pm2_rawrpc_end();
  }
}

static void service(void) {
  pm2_service_thread_create(f, NULL);
}

int pm2_main(int argc, char *argv[]) {
  pm2_rawrpc_register(&service_id, service);
  common_pre_init(&argc, argv, NULL);
  common_post_init(&argc, argv, NULL);

  if (pm2_self() == 0) {
    char msg[SIZE];
    strcpy(msg, "Init");

    tprintf("Sending string: %s\n", msg);
    pm2_rawrpc_begin(1, service_id, NULL);
    pm2_pack_byte(SEND_CHEAPER, RECV_CHEAPER, msg, SIZE);
    pm2_rawrpc_end();
  }

  common_exit(NULL);
  exit(EXIT_SUCCESS);
}
```

3.7 A minimal NewMadeleine program: Hello World!

NewMadeleine provides two API's : a send/receive interface similar to the point-to-point communication interface in MPI and a pack/unpack interface similar to the one provided by Madeleine. Programs 3.11 and 3.12 show minimal NewMadeleine programs. You can compile and execute them.

Program 3.10 Compiling and executing NewMadeleine applications

```
ravel% setenv PM2_FLAVOR nmad-mpi
ravel% cd $PM2_ROOT/nmad/examples/getting_started
ravel% make gs_pack_hello
...
<<< Generating libraries: done
    building gs_pack_hello.o
    linking gs_pack_hello
ravel% pm2-conf ravel debussy
The current PM2 configuration contains 2 host(s) :
0 : ravel
1 : debussy
ravel% pm2-load gs_pack_hello
##### ravel
##### debussy
buffer contents: <hello, world!>
ravel%
ravel%
ravel% make gs_sr_hello
...
<<< Generating libraries: done
    building gs_sr_hello.o
    linking gs_sr_hello
ravel% pm2-load gs_sr_hello
##### ravel
##### debussy
buffer contents: <hello, world!>
ravel%
```

3.7.1 Initialisation and termination

3.7.2 Communication interface

The send/receive interface

The send/receive interface mainly provides methods to send or receive data in a asynchronous way, and to test and to wait for the completion of a communication request.

The pack/unpack interface

A call to `nm_begin_packing()` initialises a sending connection. Data can then be packed into this connection using succcessive calls to the function `nm_pack()`. `nm_end_packing()` ends the building of the message and flush the data.

On the receiving side, `nm_begin_unpacking()` starts receiving and extracting a new message, data can be extracted from the current message using successive calls to `nm_unpack()` and finally, `nm_end_unpacking` ends the reception of the message.

Users have to be careful when handling data involved in a communication. Packed data should not be modified between the call to `nm_pack(...)` and `nm_end_pack(...)`, as it may modify the message contents. To overcome this restriction, a call to the function `nm_flush_packs(...)` will only return once ongoing send requests have completed. Similarly, on the receiving side, the extraction of the data may be defered until the call to `nm_end_unpacking(...)`. A call to `nm_flush_unpacks(...)` will insure previously requested data are available.

3.7.3 The MPI interface

A minimal MPI implementation has been developed on top of NewMadeleine. More informations on this implementation are available at the following URL: `http://runtime.bordeaux.inria.fr/MadMPI/`.

3.8 Leonie

So far, we used `pm2-conf` to specify which machines to run applications on, and `pm2-load` to actually start the application. These tools are written on top of and use Leonie, the PM2 bootstrap code, which reads the configuration files and then bootstraps the application. Leonie uses two configuration files:

1. *The application configuration file* which describes the machines to use for the application,
2. *The network configuration file*, as explained in Section 2.8.

3.8.1 The configuration files

Let's explain these concepts using a realistic example. Let's suppose that two clusters are available:

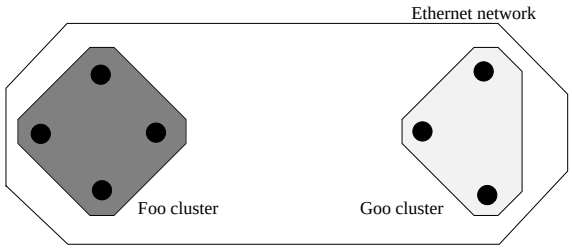

Figure 3.1: Example of interconnected clusters

- the first cluster named `foo`, is composed of 3 nodes, named `foo0`, `foo1` and `foo2`, linked by a Myrinet network;
- the second cluster named `goo`, is composed of 4 nodes, named `goo0`, `goo1`, `goo2`, `goo3`, linked by a SCI network.

Both clusters also feature an Ethernet network (TCP) which links together all the machines of each cluster. The clusters can be seen on Figure 3.1.

Leonie uses objects called *channels* in order to virtualize the available networks in a given configuration. There are basically two types of channels:

1. *physical channels* which are simple abstractions of real existing networks and;
2. *virtual channels* which are build above physical channels and can be used to create *heterogeneous networks*.

With our simple example, we can build three physical channels:

1. a channel build above the Myrinet network. This channel encompasses the nodes {`foo0`, `foo1`, `foo2`};
2. a channel build above the SCI network. This channel encompasses the nodes {`goo0`, `goo1`, `goo2`, `goo3`};
3. a channel build above the TCP network. This channel encompasses all the nodes of both clusters.

On top of these three different physical channels, we can build a virtual channel which encompasses all the nodes of the configuration. One may think that there is no difference with the TCP physical channel, but in fact the behavior of a program using the virtual channel will be totally different as Leonie will automatically select the best available network to communicate between two nodes of this virtual channel.

Indeed, all communications occurring within the `foo` cluster will use the Myrinet network, all communications occurring within the `goo` cluster will use the SCI network. And if two nodes belonging to different clusters want to exchange messages, the TCP network will be used.

More complicate configurations can be expressed: if we suppose now that the node `goo3` features a SCI NIC, we can build a virtual channel over the physical channels corresponding to the Myrinet and SCI networks. In that case, we do not need (and most important: use) the TCP network. In fact, with that new configuration, from the application's point of view, all the nodes can communicate with each other. Indeed even if a node `A` is not *physically* connected to a node `B`, it can in any case send messages to it. Internally, the node `goo3`, which features both Myrinet and SCI NICs, will **forward** the Madeleine message from `A` to `B`.

How is this information given to Leonie/Madeleine? The library uses configuration files (the following example files describe the configuration shown in Figure 3.2 with the node `goo3` featuring a SCI NIC):

- the first file, the network configuration file, named `localnet.cfg`, describes the different available networks.

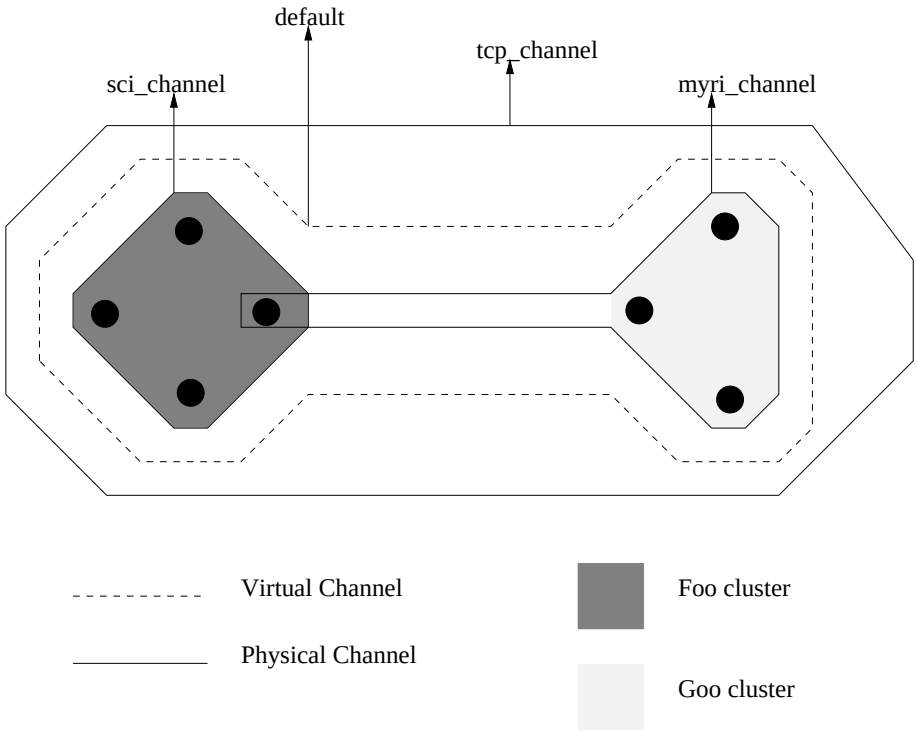

Figure 3.2: Example of configuration

```
networks : ({
             name  : tcp;
             hosts : ( foo0, foo1, foo2, goo0, goo1, goo2, goo3 );
             dev   : tcp;
           },{
             name  : myrinet;
             hosts : ( foo0, foo1, foo2, goo3 );
             dev   : mx;
           },{
             name  : sci;
             hosts : ( goo0, goo1, goo2, goo3 );
             dev   : sisci;
           });
```

A network is defined with its name (tag `name`), a list of machines it includes (tag `hosts`), and the identifier of the device connecting these machines (tag `dev`, should be one of the predefined identifiers recognized by Madeleine).

- the second file, the application configuration file, named `appli.cfg`, describes the application and the channel mapping over the different nodes.

```
application: {
  flavor : mad3;
  networks : {
    include : localnet.cfg;
    channels : ({
                    name  : tcp_channel;
                    net   : tcp;
                    hosts : ( foo0, foo1, foo2, goo0, goo1, goo2, goo3 );
                },{
                    name  : sci_channel;
                    net   : sci;
                    hosts : ( goo0, goo1, goo2, goo3 );
                },{
                    name  : myri_channel;
                    net   : myrinet;
                    hosts : ( foo0, foo1, foo2, goo3 );
                });
    vchannels : {
                    name     : default;
                    channels : ( myri_channel, sci_channel );
                 };
  };
};
```

A application is defined with a flavor (tag `flavor`), a network configuration file (tag `include`), a list of physical channel and a virtual channel.

A physical channel is defined with its name (tag `name`), the identifier of the network is based on (tag `net` which has to be defined in the network configuration file), and a list of machines it encompasses (tag `hosts`).

A virtual channel is defined with its name (tag `name`) and the list of physical channels it is build upon (tag `channels`).

Once a virtual channel is build, the physical channels below it are no longer visible by the application. However, it is possible to create several different Madeleine physical channels over the same physical network. Hence a physical channel can truly be seen as a *logical network* or a *network abstraction*.

3.8.2 Starting a PM2 application using Leonie

Once the configuration files describing your machines are written, a simple call of `leonie` to start an application would be:

```
% leonie --x --p --appli=hello appli.cfg
```

`leonie` accepts different parameters. The call `leonie --help` shows the list of all these parameters.

Program 3.11 NewMadeleine application using the pack/unpack interface

```
/*
 * NewMadeleine
 * Copyright (C) 2006 (see AUTHORS file)
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or (at
 * your option) any later version.
 *
 * This program is distributed in the hope that it will be useful, but
 * WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
 * General Public License for more details.
 */


#include <stdlib.h>
#include <stdio.h>
#include <stdint.h>
#include <nm_public.h>
#include <nm_launcher.h>
#include <nm_pack_interface.h>
#include <pm2_common.h>

const char *msg = "hello, world!";

int main(int argc, char **argv)
{
  char *buf = NULL;
  uint64_t len;
  nm_pack_cnx_t cnx;
  int            rank;
  int            peer;
  nm_core_t      p_core    = NULL;
  nm_gate_t      gate_id   = NULL;

  nm_launcher_init(&argc, argv);
  nm_launcher_get_core(&p_core);
  nm_launcher_get_rank(&rank);
  peer = 1 - rank;
  nm_launcher_get_gate(peer, &gate_id);

  len = 1+strlen(msg);
  buf = malloc((size_t)len);

  if (rank != 0) {
    /* server */
    memset(buf, 0, len);

    nm_begin_unpacking(p_core, NM_ANY_GATE, 0, &cnx);
    nm_unpack(&cnx, buf, len);
    nm_end_unpacking(&cnx);

    printf("buffer contents: <%s>\n", buf);
  }
  else {
    /* client */
    strcpy(buf, msg);

    nm_begin_packing(p_core, gate_id, 0, &cnx);
    nm_pack(&cnx, buf, len);
    nm_end_packing(&cnx);
  }

  free(buf);
```

Program 3.12 NewMadeleine application using the send/receive interface

```
/*
 * NewMadeleine
 * Copyright (C) 2006 (see AUTHORS file)
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or (at
 * your option) any later version.
 *
 * This program is distributed in the hope that it will be useful, but
 * WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
 * General Public License for more details.
 */

#include <stdlib.h>
#include <stdint.h>
#include <nm_public.h>
#include <nm_launcher.h>
#include <nm_sendrecv_interface.h>
#include <pm2_common.h>

const char *msg = "hello, world!";

int
main(int          argc,
     char       **argv) {
  char          *buf    = NULL;
  uint64_t       len;
  int            rank;
  int            peer;
  nm_core_t      p_core    = NULL;
  nm_gate_t      gate_id   = NULL;

  nm_launcher_init(&argc, argv);
  nm_launcher_get_core(&p_core);
  nm_launcher_get_rank(&rank);
  peer = 1 - rank;
  nm_launcher_get_gate(peer, &gate_id);

  len = 1+strlen(msg);
  buf = malloc((size_t)len);

  if (rank != 0) {
    nm_sr_request_t request;
    /* server
     */
    memset(buf, 0, len);

    nm_sr_irecv(p_core, NM_ANY_GATE, 0, buf, len, &request);
    nm_sr_rwait(p_core, &request);
    printf("buffer contents: <%s>\n", buf);
  }
  else {
    nm_sr_request_t request;
    /* client
     */
    strcpy(buf, msg);

    nm_sr_isend(p_core, gate_id, 0, buf, len, &request);
    nm_sr_swait(p_core, &request);
  }

  free(buf);
  common_exit(NULL);
```

Chapter 4

Customizing and debugging PM2

4.1 Introducing PM2 flavors

You have now compiled and run a few PM2 programs. It is time to have a look at the underlying machinery. The key feature is the notion of *flavor*, which we will explain in detail below. Technically speaking, a PM2 module is a library to be linked with your program. For instance, the module `pm2` is attached to the library `libpm2.a`. It is the job of the `pm2-config` utility to automatically generate the correct compiling and linking set of directives for the various modules.

```
# Assuming you use csh...
setenv CC       "`pm2-config --cc`"
setenv CFLAGS   "`pm2-config --cflags`"
setenv LIBS     "`pm2-config --libs`"
$CC $CFLAGS hello.c $LIBS -o hello
```

It is important to realize at this point that the various PM2 modules are independently compiled into *separate* libraries. Yet, these libraries are *strongly* dependent one on each other, and they are moreover dependent on the directives used in compiling the main program. Therefore, uttermost care is needed in managing the various sets of compiling and linking directives, as they *must* be consistent.

Another aspect is that many users are interested in using *several* versions of PM2 at a time. For instance, you may wish to compare the performance of a given user program on a Myrinet and on a SCI network within the same benchmarking session. Or, you may use a version of the program compiled with all the debugging features turned on, and another one aggressively optimized. You may even compare a version of PM2 compiled with some C compiler against another version compiled with another C compiler. Different decisions will result in different PM2 libraries. Changing the network from Myrinet to SCI requires a recompilation of the Madeleine module. Changing the mode from debugging to optimizing requires a total recompilation of all the libraries from scratch. This is a not trivial operation, as it consumes time and resources.

All these motivations have led to the notion of *PM2 flavors*. A PM2 flavor is simply a set of compiling and linking options for the user program and the various modules of PM2. It is always assumed that the user program and the modules have been compiled and linked with the *same* flavor. Unfortunately, there is no easy way to enforce this in the C/Unix framework. The method used in PM2 to enforce global consistency is explained in the following.

- The list of all currently available flavors is provided by the command `pm2-flavor list`.

```
ravel % pm2-flavor list
default ezflavor leonie leoparse libpthread mad3 nmad marcel
marcel-act marcel-actsmp marcel-bubblegum marcel-bubbles
marcel-mono marcel-prof marcel-smp pm2 sigmund
```

- At any point, the currently active flavor is the value of the shell variable `PM2_FLAVOR`. Alternatively, most PM2 commands provide a `--flavor=xxx` flag, so that you can easily manage several flavors at the same time within a Makefile or a script. The default of `xxx` is the value of the variable `PM2_FLAVOR`.

  Warning The `PM2_FLAVOR` variable should most preferably be set to a value! Remember to `export` it if you use `sh`-like shells.

- The flavor `xxx` is kept in the file `${PM2_HOME}/.pm2/flavors/xxx`, as indicated by the command `pm2-flavor find --flavor=xxx`.

  Warning These files are automatically generated by the tools to be presented below. Do not edit them by hand!

```
ravel% pm2-flavor find --flavor=mad3
/home/nfurmento/.pm2/flavors/mad3
ravel% more ‘pm2-flavor find --flavor=mad3‘
# Flavor mad3
... a very long file
```

- The list of options defined by the flavor `xxx` is provided by the command `pm2-flavor get --flavor=xxx`.

```
ravel% pm2-flavor get --flavor=pm2
 --flavor=pm2
 --builddir=$flavor
 --ext=
 --modules=pm2 --modules=marcel --modules=mad3 --modules=tbx
       --modules=ntbx --modules=init
 --pm2=opt --pm2=build_static
 --marcel=opt --marcel=build_static --marcel=mono
     --marcel=smp_shared_queue --marcel=marcel_main
 --mad3=opt --mad3=build_static --mad3=tcp
 --tbx=opt --tbx=build_static
 --ntbx=opt --ntbx=build_static
 --init=opt --init=build_static
 --common=opt --common=build_static --common=static
 --appli=opt --appli=build_static
```

- The internal consistency of the flavor `xxx` can be checked with the command `pm2-flavor check --flavor=xxx`.

```
ravel% pm2-flavor check --flavor=pm2
```

- A flavor may be regenerated by using the command `pm2-flavor regenerate --flavor=xxx`.

```
ravel% pm2-flavor regenerate --flavor=mad3
flavor 'mad3' unmodified
```

- When a module is compiled with the flavor `xxx`, then the resulting object code is created into the directory `${PM2_BUILD_DIR}/${PM2_ASM}/xxx`. You may thus have several versions of PM2 ready for use at the same time.
- To compile and link a user program with the flavor `xxx`, use the flags generated by the commands `pm2-config --cflags` and `pm2-config --libs`, as shown in the Makefile in Program 3.1.

4.2 Creating your own, personal PM2 flavors

You know now *what* a flavor is, and more or less *why* it is so. Let us now learn *how* they can be used. Assume for instance that you have been using the flavor `pm2` since your (PM2-)birth: `setenv PM2_FLAVOR pm2`. After having grown up, you are now starting feeling somehow unsatisfied with it, almost frustrated. No problem, PM2 has something for you: `ezflavor`! (To be pronounced: *ee-zee-flavor*)

The `ezflavor` utility is a graphic interface to list, modify, check, and regenerate flavors all in once. As it is highly dependent on the operating system you are running, you first have to compile it:

```
ravel% cd ${PM2_ROOT}/ezflavor
ravel% make FLAVOR=ezflavor
```

The binary will be created in the directory

`${PM2_BUILD_DIR}/${PM2_ASM}/ezflavor/ezflavor/bin/`

The root script to invoke `ezflavor` is available from your `PATH` in `${PM2_ROOT}/bin/ezflavor`, using your PM2 configuration, it will automatically redirect your call to the binary file.

So now, let's start `ezflavor` so to analyze your PM2-frustration.

```
ravel% ezflavor &       # Better run it in background!
```

Load in the current flavor `pm2`.

- Select the `pm2` module. The result is displayed on Figure 4.1. As you can see, only one option is turned on for this module: `opt`, which makes the module run in optimized mode.
- Select the `mad3` module: you can observe that the `tcp` option is turned on, which makes Madeleine use the TCP/IP interface.
- Select the `marcel` module: you can observe the `mono` option is turned on, which makes Marcel run on a single processor at each node (even though the node may be have multiple processors). Also, you can observe that the `marcel_main` option is on, which specifies that the `main` function of the C program is provided by PM2 (actually, by Marcel). The user only specifies the auxiliary `pm2_main` function, which is called by the real `main` function after some initialization. See for instance Program 3.5.

- Finally, you can observe that the `opt` option is enabled for the application itself: the user code is compiled with the `-O6` option of the `gcc` compiler.

Assume now that the real PM2 you are striving after is a PM2 with fully-fledged debugging facilities. What you need is to create a new flavor that we will called `pm2debug` based on the `pm2` one.

Select the `debug` option in the `pm2` module. Do it also for the application, and for the `mad3` module. Change the name of this new flavor from `pm2` to `pm2debug`, and save it (see Figure 4.2).

The new flavor is checked for consistency and saved into the file

```
${PM2_HOME}/.pm2/flavors/pm2debug
```

You may check that the file actually contains the new settings:

```
ravel% more ${PM2_HOME}/.pm2/flavors/pm2debug
# Flavor pm2debug
[...]
### SETTINGS: --pm2=build_static --pm2=debug --pm2=gdb --pm2=opt
[...]
### SETTINGS: --appli=debug --appli=opt
```

Now, you have to recompile PM2 for this new flavor. Observe that you do not have to explicitly set the `PM2_FLAVOR` shell variable to this new value. Just specify it in-line to the `make` facility.

```
ravel% cd ${PM2_ROOT}
ravel% make PM2_FLAVOR=pm2debug
[...]           # Well, a bunch of lines here...
```

Everything should run smoothly, without any warning...

Once you are done, set the `PM2_FLAVOR` variable to `pm2debug` and compile your favorite PM2 program using a "dynamic" Makefile as the one on Program 3.1. Ideally, you should not see any difference with the `pm2` run.

```
ravel% cd $PM2_ROOT/pm2/examples/simple
ravel% setenv PM2_FLAVOR pm2debug
ravel% make hello
[...]
ravel% % ls $PM2_BUILD_DIR/$PM2_ASM/$PM2_FLAVOR/examples/simple/bin/hello
/home/nfurmento/pm2/build/pm2debug/i386/examples/simple/bin/hello
ravel% pm2-conf ravel ravel
The current PM2 configuration contains 2 host(s) :
0 : ravel
1 : ravel
ravel% pm2-load hello
Hello world!
Hello world!
```

Just for fun, modify the `pm2debug` flavor so as to enable the option `isoaddr_trace` in the `pm2` module. Recompile PM2 with this modified flavor. Recompile your program, for instance the simplistic `hello` on Program 3.5 and run it. You should see a lot of rather cryptic debugging information:

```
ravel% make hello
[...]
ravel% pm2-load hello
Alloc page table(3670016)
 index 0: 0    ...
isoaddr_malloc: got 16 slots locally
mmap non-contraint => 0x3ffe0000, starting slot index = 31, slots = 16
end of isoaddr_malloc: after mmap :first slot available = 48
  Set master (31) -> 31
  Set status (31) -> 2
  Set owner (31) -> 1
  Set master (30) -> 31
[...]
isoaddr_free: added to stack cache: 0x3ffe0000 (index = 31)
Alloc page table(3670016)
 index 0: 1     ...
isoaddr_malloc: got 16 slots locally
mmap non-contraint => 0x3fff0000, starting slot index = 15, slots = 16
end of isoaddr_malloc: after mmap :first slot available = 32
  Set master (15) -> 15
  Set status (15) -> 2
  Set owner (15) -> 0
  Set master (14) -> 15
[...]
isoaddr_malloc(65536)
isoaddr_malloc: got 16 slots locally
mmap non-contraint => 0x3ffd0000, starting slot index = 47, slots = 16
end of isoaddr_malloc: after mmap :first slot available = 64
  Set master (47) -> 47
  Set status (47) -> 2
  Set owner (47) -> 0
  Set master (46) -> 47
[...]
isoaddr_free: added to stack cache: 0x3ffd0000 (index = 47)
isoaddr_free: added to stack cache: 0x3fff0000 (index = 15)
Flushing slot cache...
Flushing stack cache...
Flushing migration cache...
Isoaddr exited
Flushing slot cache...
Flushing stack cache...
Flushing migration cache...
Isoaddr exited
Hello world!
Hello world!
```

Warning Be careful that the simple Makefile of Program 3.1 is not able to spot that the flavor has been modified. You will have to explicitly delete the object file to create a fresh, consistent one.

It is often interesting to enable some options for *all* the modules at the same time, together with the application. This can be the case for example for the `debug`, `gdb`, `opt` (optimization) and `profile` options. `ezflavor` provides such a facility: click on `View/Common options/Display panel` to open the appropriate window, as shown on Figure 4.3.

A note to the wizards. Experienced and hurried users may find that the `ezflavor` interface is too heavy with respect to their needs. A tty-oriented interface is also available. It

is called `pm2-config-flavor`.

```
ravel% pm2-config-flavor
Usage: pm2-config-flavor [--text | --dialog | --xdialog] [Command ...]

Commands:
    create [flavor [model]] : create flavor 'flavor' based on model 'model'
    modify [flavor]         : modify flavor 'flavor'
    export [flavor [file]]  : export flavor 'flavor' to file 'file'
    import [flavor [file]]  : export flavor 'flavor' from file 'file'
    see    [flavor]         : show the flavor 'flavor'
    check  [flavor]         : check the flavor 'flavor'
    remove [flavor]         : remove the flavor 'flavor'
    regenerate              : regenerate all the flavors
ravel% pm2-config-flavor --text
================= Flavors management ==================
Existing flavors :
default leonie leoparse mad3
marcel marcel_act marcel_actsmp marcel_mono marcel_smp
pm2 pm2_mad3 pm2_marcel

******************************************************
You can :
 0) create a flavor
 1) modify a flavor
 2) export a flavor
 3) import a flavor
 4) see    a flavor
 5) check  a flavor
 6) remove a flavor
 7) regenerate the flavors
 8) quit this program
```

When modifying a flavor with the text version of `pm2-config-flavor`, the commands to add and remove modules and options are `add [options/modules]`, `sub [options/modules]` and `q` to quit, as shown in the following example:

```
Options for module pm2 (type help for help)
Permited s: debug gcc_instrument gdb noinline opt profile netthreads
one_vp_per_netthread realtime_netthreads isoaddr_trace build_static
build_dynamic build_both
Current selection: opt build_static
add debug gdb
Current selection: debug gdb opt build_static
sub opt
Current selection: debug gdb build_static
```

Warning In the current version of the system, no backup copy of a flavor is done prior editing. Be careful! In case, you think you have really messed up with one of your flavors, it is always possible to delete it and regenerate it from scratch.

```
ravel% pm2-config-flavor --text remove pm2debug
ravel% pm2-config-flavor --text create pm2debug pm2
Do you want to edit the new flavor pm2debug? [Y/n]
n
Creating flavor pm2debug
ravel% pm2-config-flavor --text see pm2debug
*************************************************
Current settings:
flavor name: pm2debug
builddir   : $flavor
extension  :
modules    : pm2 marcel mad3 tbx ntbx init appli
  Common options             : opt build_static static
  Module pm2    with options: opt build_static
  Module marcel with options: opt build_static mono marcel_main
  Module mad3   with options: opt build_static tcp
  Module tbx    with options: opt build_static
  Module ntbx   with options: opt build_static
  Module init   with options: opt build_static
  Module appli  with options: opt build_static
```

4.3 Debugging a PM2 program

You may have already experienced problems in designing PM2 programs. Debugging distributed, multithreaded programs is a difficult task, and only a few tools already exist to assist the programmer with this task. One of the goal of the PM2 project is to contribute to the design of such tools, but we have rather concentrated on *performance* debugging. Actually, PM2 provides powerful profiling facilities, to be introduced in Section **??**. In this section, we rather concentrate on the available tools at the level of programming.

The first step is to derive a specific flavor from your current one. It should enable the options `debug` and `gdb` for all the modules. By enabling the `debug` option, PM2 modules will carefully check consistency conditions at execution time, so that abnormal situations may be detected early. You may also wish to concentrate on memory allocations by enabling the `safe_malloc` and the `parano_malloc` options of the module `tbx`, the toolbox used by all PM2 modules. If you suspect a problem with synchronization objects, you will find useful options in the `marcel` module. If you wish to use the `valgrind` debugger on a program using `marcel`, you need to set the `valgrind` option of the `marcel` module to let `valgrind` know about `marcel` stacks.

Assuming that you are using the `pm2` flavor, create a new flavor `pm2debug` based on it as explained in Section 4.2. Set the `PM2_FLAVOR` shell variable to this value, and recompile PM2 with the new flavor.

```
ravel% setenv PM2_FLAVOR pm2debug
ravel% cd ${PM2_ROOT}
ravel% make
```

You then need to recompile the programs you wish to execute with this new flavor.

Warning Make sure that they are actually recompiled: it can be recommended to first delete the objects files by hand!

A second step is to insert additional trace outputs within your source code. Your are strongly advised to print on the `stderr` error descriptor instead of the regular `stdout` one. Also, you should always use the *thread-safe* version of `(f)printf`, called `t(f)printf`. This will prevent multiple Marcel threads to print concurrently, leading to scrambled text.

The `pm2-load` utility provides a number of options. The most useful one in a first place is `pm2-load -d`. It can be used with any flavor, but it is most rewarding when the current flavor enables the `gdb` for all the modules and for the application. If run with the `-d` option, `pm2-load` opens a `gdb` session on each (logical) node declared in the configuration, with a display on your console. Just typing `r` (for `run`) in the window will start the execution. Observe that your have to type it in *each* window! Before debugging your program at this level, you'd better downsize your code to some manageable configuration, say 3 or 4 nodes. Before starting the execution on the different nodes, you may add breakpoints and require all kind of functionalities provided by `gdb`. PM2 does not close the windows at the end of the run, so that you can inspect the final state of each node. Typing `Ctrl-D` in each window will close it and terminate the program.

Warning When running on different machines sharing the same file space, you might encounter problems to start the different graphical consoles hosting `gdb`. A solution to this problem is to set the shell variable `LEO_RSH`:

```
setenv LEO_RSH "ssh -X -n -f"
```

Note that this setting must be done in your configuration connection files such as `$HOME/.cshrc`.

Warning A note to eye-impaired Solaris users. The `gdb` facility is run within a `xterm`. In certain recent versions of Solaris, the default for the `xterm` background is a dark gray, which makes debugging *very* painful. You can change it to white by adding the following line to your `${HOME}/.Xdefaults` file:

```
xterm*background: White
```

Once it is done, reload your `.Xdefaults` file by issuing:

```
ravel% xrdb -merge < ${HOME}/.Xdefaults
```

The user should be aware that `gdb` itself is *not* aware of Marcel threads: `info threads` will only show LWPs, `thread` <num> will only let switch between LWPs, and `backtrace` will only show the backtrace of the *current* Marcel thread running in the chosen LWPs.

However, if Marcel was compiled with the gdb option, Marcel provides several gdb functions for printing thread state like the –marcel-xtop option:

```
(gdb) marcel-threads
0x3ff4fc00        user_task 43 I  1 machine nil
(gdb) marcel-printthread &__main_thread_struct
0xbffefc00             main 43 I  0 machine nil
(gdb)
```

One may switch to not-running threads' context thanks PM2-provided functions[1]:

[1] yet implemented on X86 only

```
(gdb) r
Program received signal SIGINT, Interrupt.
[Switching to Thread 16386 (LWP 25769)]
0x4013c3e7 in sched_yield () from /lib/libc.so.6
(gdb) bt
#0  0x4013c3e7 in sched_yield () from /lib/libc.so.6
#1  0x0804dc6d in idle_func (arg=0x0) at marcel_archdep.h:41
#2  0x0804973a in marcel_create (pid=0x0, attr=0x3ffe0000, func=0, arg=0x0)
    at marcel_sched.h:35
(gdb) set-ctx  &__main_thread_struct
switching to &__main_thread_struct(0x64c420)
(gdb) bt
#0  0x0804d5d8 in marcel_give_hand (blocked=0xbffefb6c)
    at source/marcel_sched.c:1173
#1  0x0804ef40 in marcel_sem_P (s=0xbffefb6c) at source/marcel_sem.c:47
#2  0x08049049 in marcel_main (argc=1, argv=0xbfffcb94) at sumtime.c:75
#3  0x08049dc1 in main (argc=1, argv=0xbfffcb94) at source/marcel.c:1276
(gdb)
```

One can then use usual functions like `frame`, `print`, etc. to inspect the thread.

The user should be aware that although `set-ctx` achieves a context switch so as to get correct `PC` and `SP`, `continue`-like commands automatically switch back to the current running thread[2]. Sometimes gdb has troubles with switching stacks, it emits a `Couldn't write registers: Input/output error.` error. In such case, one can use the three `__set-ctx1`, `__set-ctx2`, and `__set-ctx3` commands one after the other, repeating them as many times as necessary until they all worked at least once.

The user should be aware that some of the names of PM2 routine are in fact on-line definitions in internal header files. Thus, one has to know the *actual* name of the routine to set a breakpoint. The following table provides the some useful translations.

External name	Real name	Origin module
`pm2_init`	`common_init`	`pm2`
`pm2_main`	`marcel_main`	`pm2`
`tprintf`	`marcel_printf`	`tbx`
`tfprintf`	`marcel_fprintf`	`tbx`

4.4 Using the debug facilities of Leonie

The PM2 bootstrap code, `leonie`, is used by `pm2-load` to launch the application on the requested nodes. `leonie` accepts different parameters for debug purpose. It is possible to trace or log any of the modules used by PM2. The general syntax of `leonie` is:

```
% leonie [leonie parameters] configuration file [application parameters
            to be passed over to the processes]
```

A simple call of `leonie` would be:

```
% leonie --x --p --appli=hello appli.cfg
```

where the option `--x` indicates that session processes should not be started within a new graphical console (i.e. `xterm`), and the option `--p` indicates there should be no pause

[2]In theory, the PM2 process should then continue properly, but it seems gdb does not yet let this work correctly

following the termination of the session processes. Start `leonie` without these options to fully understand their behavior. The call `leonie --help` shows the list of all the available options.

Debug parameters allow to trace specific modules. The general format of a debug parameter is `--debug:<MODULE_NAME>-<TRACE_LEVEL>`. For example, the debug parameter `--debug:ntbx-trace` will display all the trace messages within the module `ntbx` either made by `leonie` or by the processes started by `leonie` (depending on the parameter is specified as a `leonie` parameter or as a application parameter). **Note that the module must have been compiled with the option** `debug`. More debug parameters are available, you can print the list as follows:

```
% leonie --x --p --appli=mad_ping appli.cfg --debug:register
(ffffffff:-99:               ) register debug name: register [default] (show=5)
(ffffffff:-99:               ) register debug name: default [default] (show=2)
(ffffffff:-99:               ) register debug name: ma [default] (show=DEFAULT (2))
(ffffffff:-99:               ) register debug name: mar-mdebug [ma] (show=DEFAULT (2))
(ffffffff:-99:               ) register debug name: marcel-init [mar-mdebug] (show=DEFAUL
(ffffffff:-99:               ) register debug name: log [default] (show=DEFAULT (2))
....

% leonie --x --p --appli=mad_ping appli.cfg --debug:mar-mdebug
(ffffffff:-99:               )                  <main_thread is bffefe00>
(ffffffff:-99:               ) Init running level 3 (Init scheduler) start
(ffffffff:-99:               ) Init running level 0 (Init self)
....
```

The `leonie` parameter `-l` indicates the output of the debug should be redirected to a file in the default temporary directory. On a typical Unix system, the name of the file will be similar to `/tmp/pm2log-$USER-x`.

When executing a Madeleine application, the flavor `leonie` is used by `leonie` itself, and the flavor `mad3` is used for the application started by `leonie`. The following command will print the list of modules for a specific flavor:

```
% pm2-config --flavor=mad3 --modules
mad3 marcel tbx ntbx init
```

The debug parameters can be specified directly for `leonie`:

```
% leonie --x --p --debug:leonie-trace --appli=mad_ping appli.cfg
% leonie --x --p --debug:ntbx-trace --appli=mad_ping appli.cfg
```

or for the processes started by `leonie`:

```
% leonie --x --p --appli=mad_ping appli.cfg --debug:mad3-log
% leonie --x --p --appli=mad_ping appli.cfg --debug:mad3-trace
%
% leonie -l --p --appli=mad_ping appli.cfg --debug:mad3-trace
% leonie -l --p --appli=mad_ping appli.cfg --debug:mad3-log
```

or for both `leonie` and the processes started by `leonie`:

```
% leonie -l --p --debug:leonie-trace --appli=mad_ping appli.cfg --debug:mad3-trace
% leonie -l --p --debug:ntbx-trace --appli=mad_ping appli.cfg --debug:ntbx-trace
```

4.4.1 Debugging PM2 processes

When starting `leonie`, you can specify processes should be started under the debugger by using the option `-d` as in the following example:

```
% leonie -d --appli=mad_ping appli.cfg
```

This command will start the processes under the GNU debugger, each within a new graphical console. If you do not have the option of starting graphical tools, you should try the following command:

```
% leonie -d --x --appli=mad_ping appli.cfg
```

If your system allows users to create core files, this command will dump the execution of the faulty processes into a core file. You can then use the GNU debugger to examine the execution in more detail.

```
% pm2-which mad_ping
/home/nfurmento/build/mad3/examples/bin/mad_ping
% gdb /home/nfurmento/build/mad3/examples/bin/mad_ping ~/core.2994
GNU gdb Red Hat Linux (6.3.0.0-1.84rh)
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu"...
Using host libthread_db library "/lib64/libthread_db.so.1".

Core was generated by '/home/nfurmento/build/mad3/examples/bin/mad_ping
      --mad_leonie node-22.'.
Program terminated with signal 11, Segmentation fault.
...
#0  0x0000000000404084 in pseudo_main (_madeleine=0x5bfef0) at mad_ping.c:820
820       session       = madeleine->session;
(gdb) bt
#0  0x0000000000404084 in pseudo_main (_madeleine=0x5bfef0) at mad_ping.c:820
#1  0x000000000044fe65 in marcel_sched_internal_create (cur=0x0, new_task=0x0,
    attr=0x0, dont_schedule=0, base_stack=0)
    at /home/nfurmento/work/pm2/marcel/include/scheduler-marcel/marcel_sched.h:436
#2  0x0000000000000000 in ?? ()
(gdb)
...
```

4.4.2 Debugging Leonie

You might need to start `leonie` itself under the debugger. To do so, you need to set the environment variable `LEO_DEBUG` to the value `1` before starting `leonie`.

```
% export LEO_DEBUG=1
% leonie --x --p --appli=mad_ping appli.cfg
GNU gdb 6.3-debian
...

(gdb)
```

The debugger then waits for some user input, you can for example set breakpoints or start the application. The file `$HOME/.leo_gdb_init` can be used to define a list of GDB

commands to execute when starting the debugger. You can for example automatically start the execution of the application.

```
% echo "r" > ~/.leo_gdb_init
% leonie --x --p --appli=mad_ping appli.cfg
GNU gdb 6.3-debian
...

##### joe
##### joe
(joe): My global rank is 1
(joe): My global rank is 0
test series completed
...
Program exited normally.
(gdb)
```

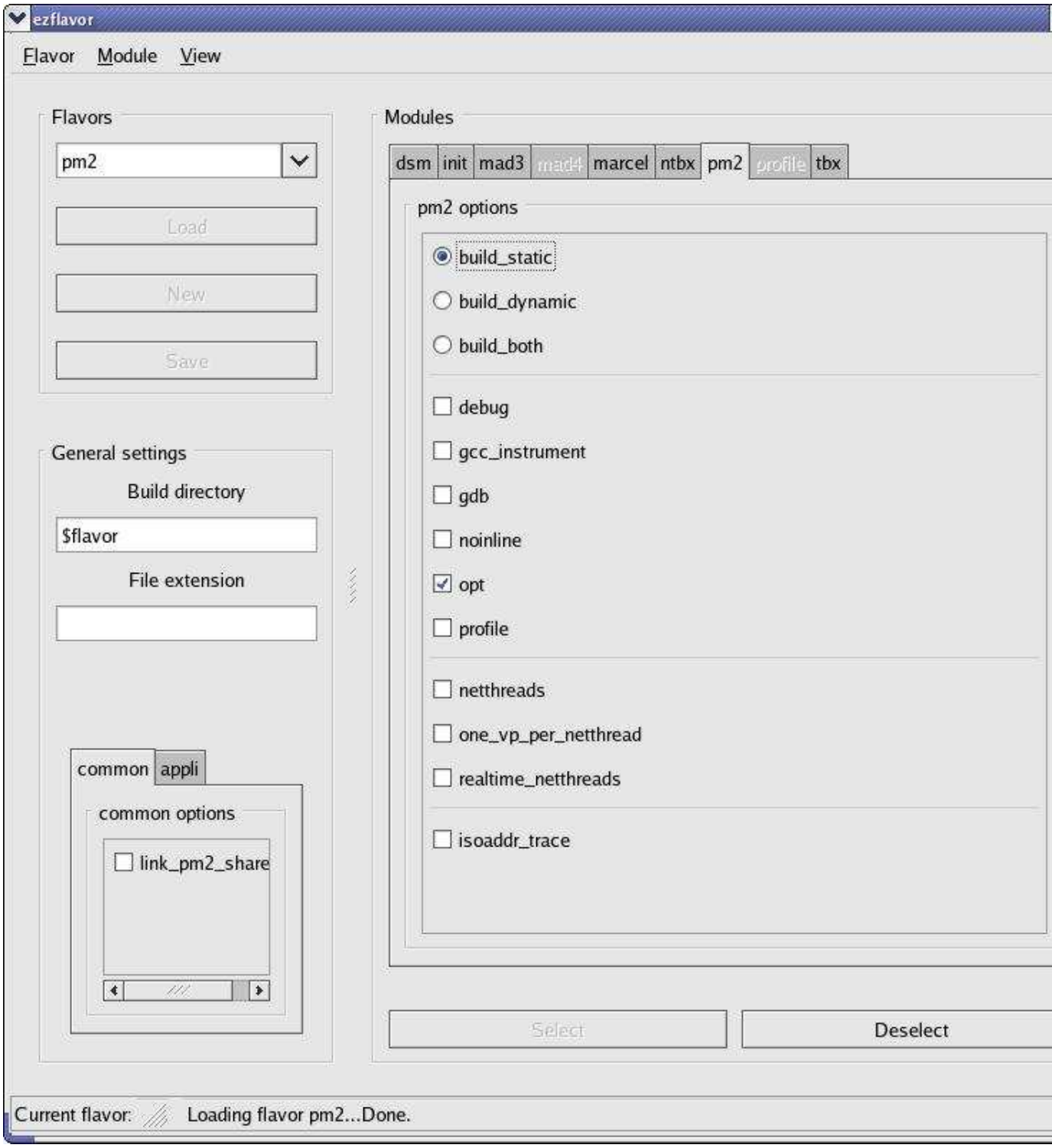

Figure 4.1: The ezflavor graphic interface to manage PM2 flavors.

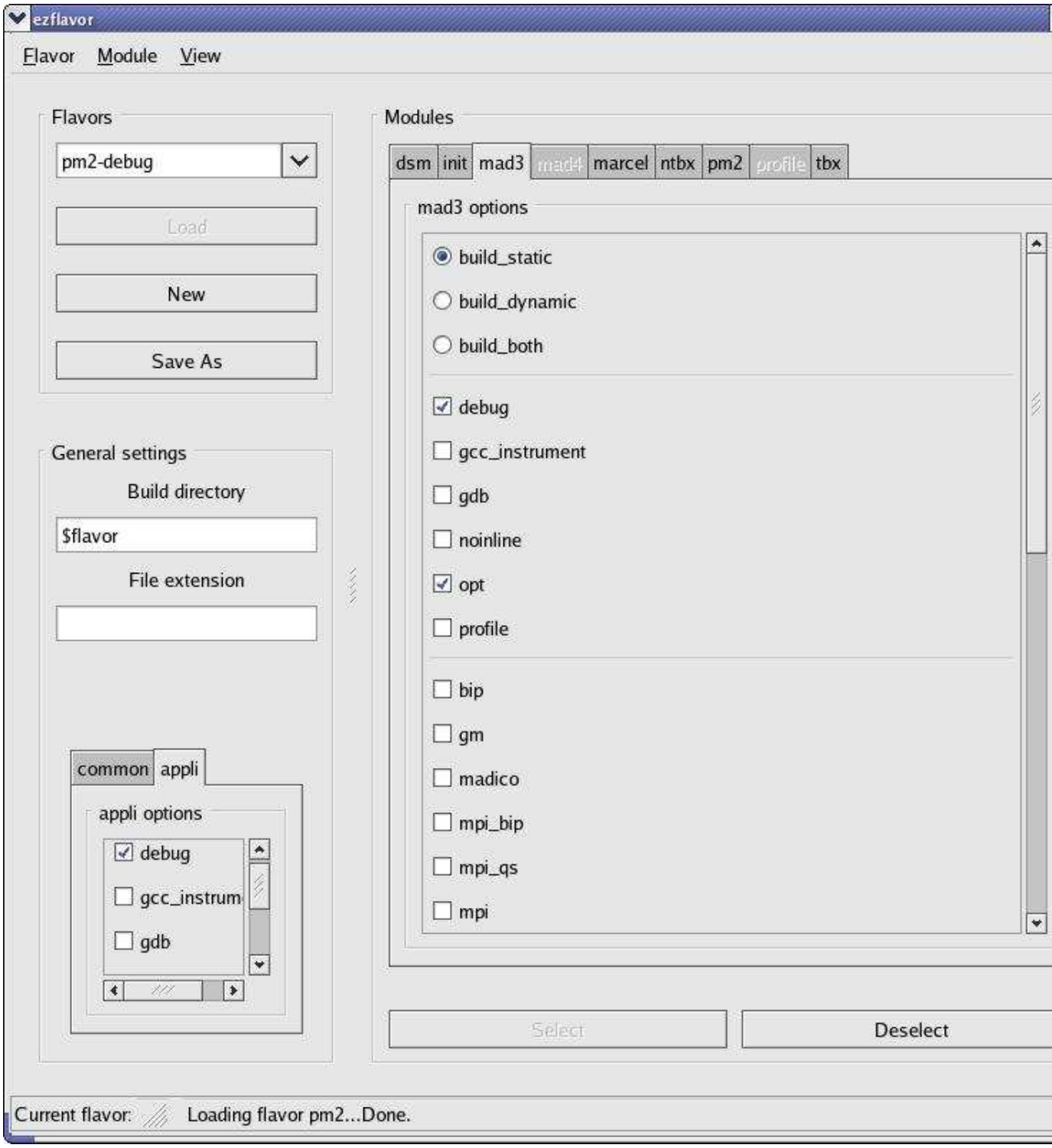

Figure 4.2: Creating a new flavor with ezflavor

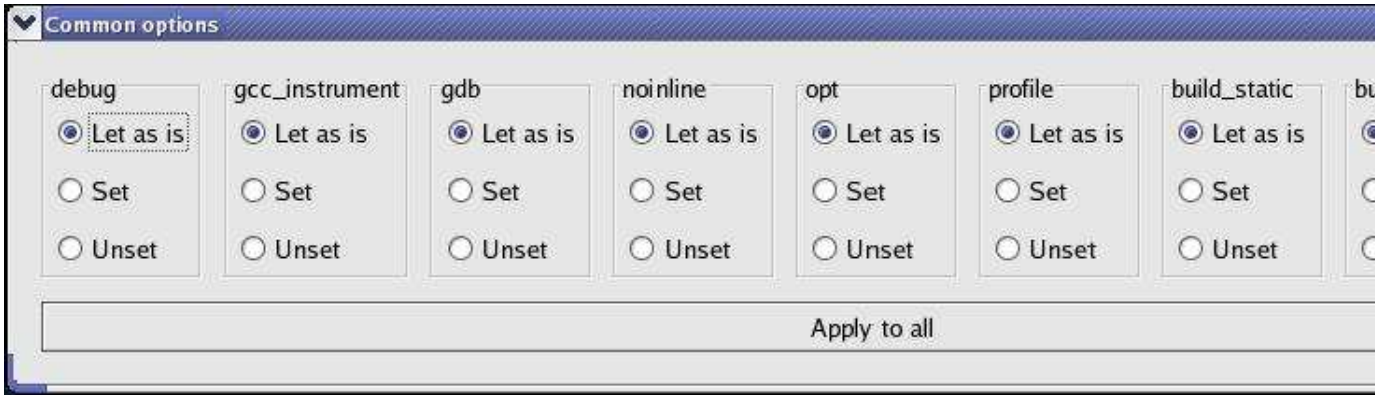

Figure 4.3: The 'Common options' window of the ezflavor utility.

Chapter 5

Mastering PM2

5.1 Mastering Marcel

Marcel has some functionalities which are disabled by default:

- The support of the `atexit` function,
- The support of blocking system calls,
- The support of the cleanup functions,
- The support of marcel exceptions,
- The support of the keys/thread-specific data handling mechanisms,
- The support of the `once` function,
- The support of the `postexit` function,
- The support of the signals mechanism,
- The support of the `suspend` function,
- The support of the userspace mechanism.

To enable them, you need to change your PM2 flavor by adding the options `enable_atexit`, `enable_blocking`, `enable_cleanup`, `enable_exceptions`, `enable_keys`, `enable_once`, `enable_postexit`, `enable_signals`, `enable_suspend`, or `enable_userspace`.

Marcel can be made to use part of the machine's processors: adding `--marcel-nvp 4` to the command line will make it use only 4 processors. By default, Marcel tries to *spread* over the machine, i.e. when using 4 processors of a quad-quad core machine, it will use one core of each of the 4 dies. To use *adjacent* processors, append `--marcel-cpustride 1` to the command line. Eventually, appending the `--marcel-firstcpu 4` option makes Marcel use processors from the fifth processor. For instance, on a quad-quad core machine, one could run 4 Marcel applications this way:

- appli1 –marcel-nvp 4 –marcel-cpustride 1 –marcel-firstcpu 0
- appli2 –marcel-nvp 4 –marcel-cpustride 1 –marcel-firstcpu 4

- appli3 –marcel-nvp 4 –marcel-cpustride 1 –marcel-firstcpu 8
- appli4 –marcel-nvp 4 –marcel-cpustride 1 –marcel-firstcpu 12

Each application will get bound to a die.

Other Marcel options are available, append `--marcel-help` to the command line to get some help.

Chapter 6

Additional PM2 material

6.1 Synopsis of PM2 scripts

6.1.1 pm2-conf

```
ravel% pm2-conf -h
Usage: pm2-conf { option } { item }
  option:
    -f <name> : Use flavor "name" (default=$PM2_FLAVOR or default)
    -p <name> : Store parameters in file "name" under the preference directory
    -h        : Display this help message
  item:
    <host>            : Add machine "host" to the configuration
    -l <file>         : Use <file> as an input for host names
    -e <host> <i>-<j> : Expand to <hosti> <hosti+1> ... <hostj>
    -s <suffix> item  : Add <suffix> to the "item" expression
```

A typical use of the extended options could thus be:

```
ravel% pm2-conf -s tcp cluster0 -s tcp -e cluster 10-12
The current PM2 configuration contains 4 host(s) :
0 : cluster0tcp
1 : cluster10tcp
2 : cluster11tcp
3 : cluster12tcp
```

6.1.2 pm2-load

```
ravel% pm2-load -h
Usage: pm2-load { option } command

  option:
    -f name : Use flavor named "name" (default=$PM2_FLAVOR or default)
    -d      : Run command in debug mode (not supported by all implementations)
    -t      : Run command through strace
    -s      : Run scripts in debug mode
    -c name : Use console named "name"
    -l      : Also generate a log file on first node
    -L      : As '-l' but do not display the log file
    -u      : Use local flavor (re-run pm2-config on each node)
    -h      : Display this help message
```

6.2 Frequently asked questions

Table 6.1 sums up the symptoms together with possible solutions.

Symptoms	**Solutions**
`Permission denied` *error at load time*	Each configuration node must be made accessible by `ssh` from the local host. Check you can issue a `ssh` call to the nodes.

Table 6.1: Common Pitfalls

Chapter 7

Reading on PM2

You can access from the following URLs a list of articles about Madeleine and NewMadeleine, the communication libraries, and Marcel, the thread library of PM2:

- `http://runtime.bordeaux.inria.fr/Publis/Keyword/MADELEINE.html`
- `http://runtime.bordeaux.inria.fr/Publis/Keyword/MARCEL.html`