Getting started with PM2

Olivier Aumage Gabriel Antoniu Luc Bougé Vincent Danjean
Raymond Namyst

Printed January 11, 2004

Contents

Chapter 1

Introducing PM2

$Id: introducing.tex,v 1.4 2001/04/18 15:13:13 bouge Exp $

PM2 is developed at LIP (*Laboratoire de l'Informatique du Parallélisme*), a research laboratory located at the ENS Lyon (*École Normale Supérieure de Lyon*) France, jointly supported by the INRIA and CNRS institutions. PM2 was originally designed at LIFL, University of Lille, France.

PM2 (*Parallel Multithreaded Machine*) is a distributed multithreaded programming environment designed to efficiently support 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 thread management subsystem of PM2 is called Marcel. It is designed to manage several hundreds of threads on each available physical processor. The PM2 programming interface provides functionalities for the management of this high degree of parallelism and for dynamic load balancing. Interesting features of PM2 include its priority driven scheduling policy, its thread migration mechanism and its ability to support various load balancing policies. PM2 has been design to provide threads as light as possible: the switching time is well under the micro-second. Yet, Marcel is completely written in portable C, but a dozen of lines of assembly code, which makes it portable across most processors and flavors of Unix systems, including Solaris and Linux. However, significant improvements have been made for Linux by introducing specific support into the operating system.

The communication subsystem of PM2 is called Madeleine. It is specifically designed to provide PM2 threads with efficient RPC facilities on modern high-performance networks. In particular, Madeleine strives at providing zero-copy communication in user-space to obtain a minimal latency and a maximal bandwidth: less than 10 μs latency and Gigabit/s bandwidth have been reported for Myrinet networks. Yet, the design has been careful enough to keep Madeleine portable across a wide range of communication interfaces, including TCP and MPI. This makes PM2 portable on most Unix platforms.

1.1 The PM2 Programming model

PM2 adheres to the SPMD (*Single Program Multiple Data*) programming model, in a way very similar to the PVM and MPI communication libraries. The user writes a single program text, a copy of which is launched by a specific **load** command on each *processing node* of the current configuration. Then, it is up to this common program text to include branching so as to differentiate between the processing nodes, based on a programming scheme like

```
if(pm2_self() == 0) { /* Do something... */ }
else { /* Do something else... */ }
```

At this level of presentation, a processing node is simply a Unix process. The association between *processing nodes* and *physical SMP nodes*, or even *physical processors of the SMP nodes* are made outside of the PM2 model by specific commands described below.

On each node, the **pm2_main** function of the program is activated. This function may then *spawn* (i.e., create) an arbitrary number of children threads. PM2 adheres as much as possible to the POSIX Threads programming model. A PM2 thread is just declared as an ordinary function. The only difference is that it is called through a certain library routine, which makes it run *concurrently* to its caller function. The caller and the callee share all the global variables of the local copy of program. Yet, the callee is launched with a fresh stack. This implies that there is no return for a function launched as a thread. On returning from the initial activation of its function, the thread merely dies. Observe also that the caller and the callee functions run concurrently: once launched, there is no notion of a *father* any longer, nor a *son*, and the set of threads is just flat: if one of the running threads spawns a new thread, then it is just added to the current, flat pool of running threads, whatever its creator. Also, any thread may terminate the program, whatever its rank of creation.

1.2 Threads and RPCs

Once spawned, a thread runs until its function returns. Threads living on the same node may synchronize by a number of means, as defined by the POSIX Threads model: semaphore, monitors, etc. Threads living on distinct nodes may not directly interact together. The only way a PM2 thread may interact with a remote node is to issue a *remote procedure call* (RPC): the *client* thread requires the execution of a *service* function on the remote, *server* node. PM2 provides the programmer with many flavors of RPCs. The client thread may or may not wait for the result of the function; the argument of the service function may be packed and unpacked in various ways, depending on their use; the server node may or may not create a new, fresh thread to execute the service function. This latter alternative deserves some comments. In many thread package, the default behavior is that a fresh thread is spawn on the server node to execute the service function, so that the node is left available to handle further requests. However, such a behavior induces an additional overhead which may be prohibitive with respect to the requested service. In PM2, the server node is left to execute the service function *in person* so as to save this overhead. As a side-effect, this keeps the node busy and unable to serve any further request: it is up to the programmer to make sure that no deadlock may result. Of course, service functions may also dynamically spawn threads if necessary, getting back to the usual behavior.

Synchronization between threads is a crucial aspect of multithreaded programming. If the threads live on the same node (that is, within the same Unix process), then threads can synchronize through *locks*, according to the POSIX Threads model. If they lives on distinct nodes, then PM2 defines special objects called *completions*. In short, a completion is a special, self-described token which can be generated by a thread, and then forwarded to some other threads. The originating thread can then **`wait`** for the completion to be sent back to him using a special **`signal`** routine. This routine may be executed by any thread, including a thread living at the same node as the originator.

1.3 Starting and stopping programs

How does a PM2 SPMD distributed program start and stop? This is a difficult problem, which requires much attention from the programmer as it is the never-ending source of subtle failures (the writer has painfully experienced it!). The execution of a PM2 program can be divided in three phases.

Prelude. The prelude starts on launching the **`pm2_main`** root function of the program. This function should include all necessary initializations, especially the registration of the service functions which may be activated by remote clients (**`pm2_rawrpc_register`** routine), and more generally all objects which may be seen from remote nodes. This phase ends with a call to the **`pm2_init`** routine, which spawns the servicing threads to listen to external requests (RPC, migration, etc.).

Main part. After returning from the **`pm2_init`** routine, all PM2 nodes can freely proceed, behaving both as a client and a server. In general, it is considered as *literate PM2 programming* to let only node 0 to be active, all the other nodes waiting for incoming requests.

Postlude. Each PM2 node should eventually issue a call to the **`pm2_exit`** routine. This function waits for *all* local threads to terminate, including the local internal system threads spawned by the **`pm2_init`** function to serve the remote RPC requests. Again, in *literate PM2 programming*, all nodes but node 0 should immediately call the **`pm2_exit`** function after the **`pm2_init`**, just waiting for incoming requests through internal service threads. A call to **`pm2_exit`** should be issued *exactly once by each node*: unpredictable behavior may result from multiple calls to this routine at a single node. However, any thread of the node may call it: there is no reason why this should necessarily be done by the initial thread.

Of course, such a scheme does not fully meet our needs: the system eventually reaches a state where only one thread remains living at each node, blocked within a **`pm2_exit`** call, all the service threads waiting for incoming requests. So to speak, the program has terminated, but nobody knows it for sure, so that it may not stop! This situation is well-known in the field of distributed algorithms: it has been called *distributed termination*. PM2 solves the problem by providing the programmer with a special routine called **`pm2_halt`**. This routine should be called *exactly one within all the nodes*. By issuing such a call, the programmer commits that no request remains in transit and that no further request shall ever be issued. The **`pm2_halt`** routine broadcasts a signal to all the nodes of the configuration, whose effect is to stop all the system service threads. Then, the **`pm2_exit`** routine can finally return at each node, and the local copy of the program eventually exit. Again, issuing multiple calls to the **`pm2_halt`** routine, or issuing it before all requests have been handled, will most probably lead to a deadlock.

NB *Luc to Olivier: Please, check the above section *very* carefully!*

1.4 Migrating threads

An interesting feature of PM2 is that it provides a mechanism for *preemptive thread migration*. The principle is as follows. A any point, a thread can request from the system the list of the threads living on the same node, pick up a thread (including itself!) for migration to some remote processing node and trigger its effective migration. On such a request, the system suspends the selected thread and removes it from the scheduler queue. The thread is marshaled into a buffer (i.e., its memory image is copied), and physically deleted at the node. A RPC is then issued to the remote node which has been specified as destination for the migration. The service function takes the buffer as its argument. On the destination node, this function unmarshals the buffer, installs the thread and finally inserts it into the scheduler queue, ready to run.

Seen from the thread's point of view, the migration is completely transparent, very much like a context switch. The only difference is that only the *proper resources* of the thread have been migrated: its descriptors, its stack and the memory areas which have been allocated through a specialized *iso-allocation* routine called **`pm2_isomalloc`**. Thanks to this specialized *iso-allocation* facility, all the thread's resources are relocated at the *same virtual addresses* upon a migration, so that pointers remain valid. Yet, no provision is made at this level for the data shared by the thread with other local threads, for instance the global data of the program image running at the node. It is up to the programmer to make sure that the resulting behavior is correct.

The concept of migration has been introduced above for one single thread. In fact, PM2 provides a general migration facility which enables the programmer to migrate *several threads* at a time, each thread being migrated to a specific processing node. Also, it should be stressed that the series of operations described above at the origin node can be made atomic if desired, using specific PM2 routines.

PM2 features an additional functionality to provide threads with a uniform access to data, whatever their physical location. It is called DSM-PM2. The idea is that the programmer can register certain global data in to be *uniformly shared* by all the threads. This can be done *statically* by using a special declaration for the global variables of the program, or *dynamically* by allocating memory areas with a specific allocation function. Exactly like the **`pm2_isomalloc`** function above, the system guarantees that all pointers remain valid on all nodes, so that these memory areas can be used completely transparently by the programmer. DSM-PM2 is parameterized by the consistency protocol to be used for managing shared data. A number of built-in protocols are provided in the standard distribution for various consistency models, including *sequential consistency* ones (for instance, Li-Hudak), and *release consistency*, with several alternative protocols for each of them. The protocol to be used is actually specified within the prelude phase (before the call to **`pm2_init`**) using a special routine called **`dsm_set_default_protocol`**. The user may thus switch from one protocol to another without recompiling the program! Also, additional protocols can be dynamically constructed by the user using the basic blocks provided by the library. A typical protocol may be specified at this level within a few hundreds of lines.

Chapter 2

Installing PM2

$Id: installing.tex,v 1.4 2001/04/18 15:13:13 bouge Exp $

This section describe 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.

NB *Luc: There should be something alike for Madeleine.* **`${PM2_ROOT}/INTERFACES`***?*

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

PM2 was initially designed with a preliminary version of the Madeleine. This version is now obsolete and not maintained any longer; it is referred to as *Madeleine I*. The current version of the interface is called *Madeleine II*. The internal design is completely different, and the new version is only partly compatible with the preliminary one. This manual only introduces the Madeleine II communication interface, simply referred to as *Madeleine* hereafter when no confusion may occur.

Madeleine (that is, Madeleine II) 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 as been designed to be easily portable and only relies on the availability of the two following development tools:

- GNU C Compiler **`gcc`** (versions **`2.7.2.3`**, **`2.8.1`** and higher, or **`Egcs`** versions);
- GNU **`make`** (versions **`3.7x.x`**).

Also, PM2 includes a large number of shell scripts, which are assumed to be run with the *GNU utilities*. This is specially important for **`expr`**, **`tr`**, **`head`**, **`cut`**, **`test`**, etc. (version 2.0 or higher), **`grep`** (version 2.3), **`tar`**, etc.

> **Warning** *If you encounter installation problems, first make sure that you actually use the right version of these utilities!*

NB *Luc: There should be a way of listing all utilities used in the scripts.*

NB *Luc: I do not understand what* **`Egcs`** *means here. Olivier, please correct.*

2.2 Getting the PM2 software and help

The PM2 archive containing all the necessary sources is available from the PM2 web site at:

http://www.pm2.org/

Please, send an email to pm2-users@ens-lyon.fr if you encounter problems with the PM2 software. However, please make sure first that your problem is not already listed in the *Frequently Asked Questions* Section 7.4.

2.3 Unpacking the PM2 distribution

The PM2 archive can be extracted with one of the following commands entered at the shell prompt:

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

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

bin	PM2 scripts
common	???
console	???
doc	Documentation files
dsm	Distributed shared memory manager
ezflavor	Flavor configuration graphic interface
generic	???
leonie	???
leoparse	???
mad2	Communication subsystem
make	PM2 makefiles
marcel	Multithreading subsystem
modules	???
ntbx	Generic network management utility toolbox
pm2	PM2 *en personne*!
pm2debug	PM2 debugging utilities (useful!)
profile	PM2 profiling utilities
sigmund	???
swann	???
tbx	Generic PM2 utility toolbox

The **mad1** directory contains the obsolete Madeleine I communication subsystem for backward reference, but should not be used any longer. Just ignore it.

NB *Luc to Olivier: Please fill up the holes!*

2.4 Setting the environment variables

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

- The **PM2_HOME** variable must contain the path to the PM2 administrative directory. The default is your home directory **${HOME}**. There, a **.pm2** directory will be created to store various PM2 configuration files such as flavors, etc. and some system-dependent binary files such as **ezflavor**, etc.
- The **PM2_ROOT** variable must contain the path to the PM2 distribution root directory. Observe that PM2 does not write anything into this directory: a write-protected directory is OK.
- The **${PM2_ROOT}/bin** directory has to be included in the active search path.
- Finally, all the files generated within the compilation process will be placed by default in the **${PM2_HOME}/build** directory. Though this may be convenient in a first place, you may wish to keep these files in a more dedicated place. Just define the **PM2_BUILD_DIR** shell variable to the desired value. Just note that this directory has to be reachable from any machine used in running PM2 programs. Be careful that **/tmp** does not work in general!

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
setenv PM2_BUILD_DIR     ${PM2_HOME}/build
```

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

2.5 Creating the standard PM2 flavors

You should now create your own, private configuration database. It is stored in the **${PM2_HOME}/.pm2** file in your home directory. It is made of a list of *flavors*. A flavor is simply a (rather complex!) series of options to be passed to the PM2 modules at compile-time and/or at run-time. Flavors are organized so as to let you easily maintain several versions of the PM2 system at the same time: for instance, one compiled for the TCP protocol and one compiled for the BIP/Myrinet protocol. Or, one compiled with the debugging checks on, and one with the checks 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 extracting the option sets for each PM2 module if necessary. Then, this creates the the various flavors. You can just list them by

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

In the following, we will advise you to use the **pm2** standard flavor. Just have a look at it to get a feeling of your current flavor. You'll probably never dare do it again!

```
ravel% pm2-flavor get --flavor=pm2
```

There are many more way of using the **pm2-flavor** command. See **pm2-flavor -h** for the extensive on-line help.

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.

The only interesting setting at this time is the underlying network protocol selection, which is currently not desirable to be auto-detected by the Madeleine II communication library of PM2. It is set to TCP by default in the **pm2** flavor, which should fit your needs at least for the first runs.

```
ravel% setenv PM2_FLAVOR pm2
```

2.7 Compiling the library

You are now ready for the big thing: compiling the whole library. No files are created in the **${PM2_ROOT}** directory, so that you can compile and install PM2 from a CDROM. All executable will be placed into the **${PM2_BUILD_DIR}** directory.

> **Warning** *Remember that this directory has to be accessible from all machines included in the PM2 configuration. A local* **/tmp** *directory will not work!*

To launch the whole compilation, just invoke **make**.

> **Warning** *Make sure you use the* right *version of the program (see above for detailed specification). Only recent GNU versions will make it!*

```
ravel% cd ${PM2_ROOT}/pm2
ravel% make
```

You may wish to precisely follow the compilation process, just set the **`VERB`** variable of the makefile to **`verbose`**.

```
ravel% make VERB=verbose
```

2.8 Cleaning out everything

The directory tree of the PM2 distribution may be cleaned at any time from compiled objects and libraries using the command **`pm2clean`**. This may be needed to free some disk space when PM2 is not in use or to force the whole PM2 distribution to be completely remade by a subsequent **`make`** call.

```
ravel% cd ${PM2_ROOT}/pm2
ravel% make clean
```

Note that this does not affect your private **`.pm2`** flavor file, nor your **`build`** directory. If wanted, they have to be removed by hand.

NB *Luc: Is this still correct?*

2.9 Solving problems

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

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

This result 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 restore a fresh and safe configuration for you, ready to restart the whole compilation. The output of the command may 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!)

2.10 Testing your first PM2 program

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

```
ravel% echo ${PM2_FLAVOR}
```

The PM2 library should now fully configured. You are ready to compile the example programs of the distribution. Let's start with the simplest one: *Hello world!*, located in file **hello.c**.

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

The compiled version of our program is automatically placed into your private **build/${PM2_FLAVOR}** directory, but you need not bother with this detail at this point: PM2 cares about it for you! Use the **pm2which** command to learn where a file is physically located;

```
ravel% pm2which hello
```

2.11 Specifying the configuration for execution

The final step before execution is to specify the list of host names on which the PM2 application is going to run. For this, use the **pm2conf** command. My current machine is called **ravel**, and the two neighboring ones are called **debussy** and **faure** (well, this should actually read *Fauré*!).

```
ravel% pm2conf ravel debussy faure
The current PM2 configuration contains 3 host(s) :
0 : ravel
1 : debussy
2 : faure
```

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 here my current workstation as node 0. This is for convenience only, but may machine could make it. Indeed, there is no reason why my local machine should get enrolled in the execution of the program. In running a PM2 on a dedicated cluster, the local machine is most preferably out of the configuration.

> **Warning** *Just to avoid disturbing problems in the sequel, please make sure that you local machine is allowed to issue* **`rsh`** *commands to each machine in the configuration, and that the directory* **`${PM2_ROOT}`** *is actually shared by all of them. For instance, try:*
>
> ```
> ravel% pm2-all 'ls ${PM2_ROOT}'
> ```
>
> *(Note the* `'` *characters!) This script issues a rsh command to each node in the current configuration. If you do not get the list of the files in your current* **`${PM2_ROOT}`** *directory, that is, either you get nothing or a message like* **`permission denied`***, etc., then you are in trouble!*

If **`rsh`** does not work because of right permissions, a possible turnaround is to use **`ssh`** instead of **`rsh`**:

```
ravel% setenv PM2_RSH ssh
```

If it does not output the right list of files, then the remote machines in the configuration are not correctly served by the network fil system. PM2 cannot work, as it implicitely assumes that the excutable of the program will be found on the remote machines at the same place as it is on the local one.

If this is not possible to correct such problems, then it is still possible to run all three PM2 *virtual nodes* as three processes of your local machine:

```
ravel% pm2conf ravel ravel ravel
The current PM2 configuration contains 3 host(s) :
0 : ravel
1 : ravel
2 : ravel
```

2.12 Loading the program

It remains to actually load and run the program.

```
ravel% pm2load hello
[Threads : 2 created, 0 imported (0 cached)]
Hello world!
```

Congratulations, you've run your first PM2 program! **`;-)`**

You can see that our program generates two messages. The first one is the expected *Hello World!* message coming from the master process with a call to the **`tprintf`** PM2 routine. It is a simple wrapper of the regular **`printf`** Unix routine, to be used in multithreaded context. The second one is some PM2 running statistics related to the master process. Note that the output of the nodes different from node 0 does not show up at the terminal. We will see later how to get them.

2.13 If everything else fails...

Feedback from PM2 users showed that several common pitfalls may be encountered at this time. If this is your case, please refer to Section 7.3: *Common Pitfalls*.

Chapter 3

Discovering PM2

$Id: discovering.tex,v 1.4 2001/04/18 15:13:12 bouge Exp $

This chapter aims at illustrating the basic features of PM2 through some very simple programs. 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. The only point is to include the necessary definitions and libraries. As the command line is rather complex, PM2 provides a simple utility to generate it on-line: **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 Figure 3.1 a **Makefile** which does all the work for you. Executing **make hello** will compile source file **hello.c** with all the necessary parameters.

Observe that using dynamic calls to the **pm2config** 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.

```
## 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=%)

all: $(EXEFILES)

$(EXEFILES): %: %.c
	@echo "Compiling $< for flavor '$(PM2_FLAVOR)'..."
	@$(CC) $(CFLAGS) $< $(LDFLAGS) -o $@

clean:
	-rm $(EXEFILES) $(OFILES)
```

Program 3.1: [**`Makefile1`**] A minimal GNU Makefile for PM2 programs.

3.2 Hello World!

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 PM2 programming interface.

3.2.1 The minimal PM2 program

Figure 3.2 shows an example of a minimal PM2 code. Compile it and run it:

```
ravel% make hello
[...]   # Normally, no warning whatsoever!

ravel% pm2conf ravel debussy faure
The current PM2 configuration contains 3 host(s) :
0 : ravel
1 : debussy
2 : faure

ravel% pm2load hello
[Threads : 2 created, 0 imported (0 cached)]
Hello World!
```

```c
#include <pm2.h>

int
pm2_main (int argc, char *argv[])
{
  pm2_init (&argc, argv);

  tprintf ("Hello World!\n");

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

  pm2_exit ();
  return 0;
}
```

Program 3.2: [**`hello.c`**] Minimal PM2 program.

Though it looks simple, it is a full-fledged SPMD parallel program that spawns on several processing nodes during execution! Let's examine it step by step.

Header files

A PM2 program must always include the PM2-specific **`pm2.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.

The `pm2_main` function

The main function of the PM2 program is named **`pm2_main`**, in contrast to traditional C programs that 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 7.1 for details.

PM2 initialization

A PM2 program has to call **`pm2_init`** to effectively initialize the PM2 runtime system. Moreover, this step involves a global synchronization phase (actually, a global synchronization barrier) among all the PM2 nodes so that each process is assigned a unique *rank number* on returning from **pm2_init**. 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 3.2.1 for details.

Most importantly, the **pm2_init** function spawns a number of internal *thread daemons* that are in charge of listening 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 is placed just after **pm2_init**, 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* **pm2_init**.

Note also that the Unix standard input/output streams may not be correctly initialized before the call to **pm2_init**. Thus, the behavior of programs using I/O operations before **pm2_init** is not defined.

Going back to our example program, note the call to the **tprintf** routine of PM2. This is a simple wrapper of the regular **printf** routine of the C library, which is protected against multithreading (technically, it is made *reentrant* by disabling scheduling). As mentioned previously, all nodes perform this call and actually produce the "*Hello World!*" string on their output stream. Yet, only the output of node 0 is observed here. Section 3.2.2 describes where these outputs actually go during execution.

Who's who?

Each processing node (that is, Unix process) taking part to a given execution receives its own unique *rank* number. This is an **unsigned int** between 0 and (**pm2_config_size ()** − 1). A node can learn 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 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 of use of **pm2_self**, the program features a test on the rank of the current process: if the current process is the main process, then the **pm2_halt** routine is called.

How to terminate?

The termination of a PM2 program is a delicate task since PM2 performs no automatic termination detection. Thus, the termination decision must be made at the application level. The termination phase is split in two steps.

Exactly one node must perform an explicit call to **pm2_halt**. In our example, the main process was chosen to execute this function. Actually, any process could have been selected.

For instance, we could replace

```
if(pm2_self () == 0)
```

by any other kind of test selecting exactly one node, for instance:

```
if(pm2_self () == pm2_config_size () - 1)
```

At least conceptually, the **pm2_halt** routine 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*!

To exit from a PM2 session, each node must call the **pm2_exit** routine. This call blocks the calling thread until all other threads (belonging to the same node) terminate. In our example, no other application thread but the main one is running. Thus, **pm2_exit** completes as soon as the internal daemon threads are stopped. Had we removed the call to **pm2_halt**, then all nodes would remain hanging, waiting for ever for external requests!

As the call to **pm2_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 **pm2_exit**, so that the order may in fact be irrelevant.

From the user point of view, the PM2 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 remains running after the main node has completed. In our example, the main node can even terminate before any other node has started executing **tprintf**! Fortunately, this is not a problem here, since **tprintf** does not involve any communication with other nodes. We will further discuss this termination problem in Section 3.2.2.

NB *LB: Is there a specific ordering to call halt and exit? In this section, I suggest that it is not relevant!*

3.2.2 Where does the output go?

This simple **hello** program only printed the message generated by the master node node. Indeed, the standard output of the other nodes is redirected to local log files located into the **/tmp** directory. The logs are easily accessible using the command **pm2logs** which is in charge of retrieving and displaying logs from each slave node of the session configuration. Observe that such requests are quite slow, as they use **rsh** connections:

```
ravel% pm2logs
*** Process 1 on debussy:
[Threads : 2 created, 0 imported (0 cached)]
```

```
#include <pm2.h>

int
pm2_main (int argc, char *argv[])
{
  pm2_init (&argc, argv);

  if (pm2_self () == 1)
    tprintf ("Hello World from node 1!\n");

  if (pm2_self () == 2)
    pm2_halt ();

  pm2_exit ();
  return 0;
}
```

Program 3.3: [**hello1.c**] Output redirection, Example 1.

```
Hello World!
*** Process 2 on faure:
[Threads : 2 created, 0 imported (0 cached)]
Hello World!
```

Observe that there is no reason why all the nodes should make the *same* output requests. Consider for instance the variant of the **hello** program on Figure 3.3. Only node 1 outputs a message. Also, just to show you that any node can force termination, node 2 is this time in charge of it!

```
ravel% make hello1
[...]

ravel% pm2load hello1
[Threads : 2 created, 0 imported (0 cached)]

ravel% pm2logs
*** Host debussy, process 1:
[Threads : 2 created, 0 imported (0 cached)]
Hello World from node 1!
*** Host faure, process 2:
[Threads : 2 created, 0 imported (0 cached)]
```

Redirecting output to the terminal

For user convenience, it is possible to redirect output directly to the output of the master node using the **pm2_printf** function. As **tprintf**, it is protected against multithreading. Moreover, it sends its output as a regular message to node 0, where it is printed out. How-

```
#include <pm2.h>

int
pm2_main (int argc, char *argv[])
{
  pm2_init (&argc, argv);

  if (pm2_self () == 1)
    {
      pm2_printf ("Hello World !\n");
      pm2_halt ();
    }

  pm2_exit ();
  return 0;
}
```

Program 3.4: [**`hello2.c`**] Output redirection, Example 2.

ever, the user must be aware that this facility should be used with caution, as it uses the common communication subsystem.

Consider program **`hello2`** on Figure 3.4. The output is immediately printed on the terminal attached to the master node. The **`[t1]`** label is inserted to identify the outputting node. By convention, outputs from node 0 have no label, so as to mimic the behavior of the original **`printf`** function.

```
ravel% pm2load hello2
[t1] Hello World !
[Threads : 2 created, 0 imported (0 cached)]
```

3.3 Remote Procedure Calls

The goal of this section is to let you issue a *Remote Procedure Call* in PM2. We start with a basic scheme, and we refine it step by step so as to demonstrate the versatility of PM2.

3.3.1 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 this is fully transparent for the user of PM2. Nodes are specified through their numbers, as allocated by the **`pm2conf`** command. In this part, we consider our usual 3-node configuration:

```
ravel% pm2conf ravel debussy faure
The current PM2 configuration contains 3 host(s) :
```

```
#include <pm2.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);
  pm2_init (&argc, argv);

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

      pm2_halt ();
    }

  pm2_exit ();
  return 0;
}
```

Program 3.5: [**`rpc.c`**] Defining a service in PM2.

```
0 : ravel
1 : debussy
2 : faure
```

In Program 3.5, 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_init`** routine. Doing otherwise will result in unspecified behavior, probably depending on the relative speed of the nodes. The user is therefore *strongly advised* to pay very careful attention to this requirement.

Returning to the program, processing node 0 invokes the remote service **`service_id`** on 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% pm2load rpc
[Threads : 2 created, 0 imported (0 cached)]

ravel% pm2logs
*** Process 1 on debussy:
[Threads : 2 created, 0 imported (0 cached)]
Hello, World!
*** Process 2 on faure:
[Threads : 2 created, 0 imported (0 cached)]
```

Observe that an image of the program has been started *both* on hosts **`debussy`** and **`faure`**. In fact, the service function has been registered on all 3 nodes, including root node 0. Yet, only node 1 has been used as a server for the request of node 0.

In this example, the two processing nodes of PM2 have been launched on two distinct machines for the sake of clarity. However, the machine running the server node may well be the same as the one running the client node:

```
ravel% pm2conf debussy debussy debussy
```

```
The current PM2 configuration contains 3 host(s) :
0 : debussy
1 : debussy
2 : debussy

ravel% pm2load rpc
[Threads : 2 created, 0 imported (0 cached)]

ravel% pm2logs
*** Host debussy, process 1:
[Threads : 2 created, 0 imported (0 cached)]
Hello, World!
*** Host debussy, process 2:
[Threads : 2 created, 0 imported (0 cached)]
```

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 on this in the next section!

3.3.2 Passing parameters

Let us now the case of a service function invoked with additional parameters. In Program 3.6, 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, *À 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 message 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 of any 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.

```
#include <pm2.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);
  pm2_init (&argc, argv);
  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 ();
    }
  pm2_exit ();

  return 0;
}
```

Program 3.6: [**rpc-params.c**] Service with a string parameter.

Let us now run the program.

```
ravel% pm2load rpc-params
[Threads : 2 created, 0 imported (0 cached)]

ravel% pm2logs
*** Process 1 on debussy:
[Threads : 2 created, 0 imported (0 cached)]
The sentence is: A la recherche du temps perdu.
*** Process 2 on faure:
[Threads : 2 created, 0 imported (0 cached)]
```

> **Warning** *Program 3.6 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 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__;
> ```

3.3.3 The packing/unpacking API

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 save copies: actually, the time for copying a buffer within a node is of the same order of magnitude as the time for sending it on 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. As the size of the arguments for a RPC call are usually unknown in advance, the communication interface has to include this feature into its design. The approach of PM2, or more accurately of the underlying communication library called Madeleine, is to control the packing and unpacking operations with additional *flags*.

A 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 this need. 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 semantics of the operation. Available sending flags are defined as follows:

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.

SEND_LATER This flag indicates that PM2 should not consider accessing the value of the corresponding data until the **pm2_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.

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 the **send_CHEAPER** semantics.

The following flags control the reception of user data packets:

RECV_EXPRESS This flag forces PM2 to guarantee that the corresponding data are immediately available after the the *unpacking* operation. Typically, this flag is mandatory 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 latency and bandwidth. The user should therefore extract data this way only when necessary.

RECV_CHEAPER This flag allows PM2 to defer the extraction of the corresponding data until the execution of the **pm2_rawrpc_waitdata** 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 **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 3.6, everything is sent **CHEAPER**, so as to save time. The user has been careful enough not to corrupt variables **len** and **s** before calling **pm2_rawrpc_end**. On the reception side, variable **len** *must* be received **EXPRESS**, as its value is needed to allocate the buffer for the unpacking of variable **s**. Then, variable **s** can be received **CHEAPER**. If the underlying operating system and 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 performance.

3.3.4 Threaded services

Services 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.

Such problems can be addressed by executing services into a fresh threads instead of the original service handler. This technique is demonstrated on Program 3.7. 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 do 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.

NB *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 for the sake 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 in 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 broadcasting a termination request to all the nodes. Assume now that the messages used on 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 worse case is quite common in this matter, in opposite (?) to other domain of life. Just don't try!

How can one circumvent the problem? A generic technique called *completion* will be introduced later. In the specific example displayed on Figure 3.7, 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. Observe that 3 threads have been created on node 1, instead of the regular 2.

```
#include "pm2.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);

  pm2_init (&argc, argv);

  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!!! */
    }

  pm2_exit ();

  return 0;
}
```

Program 3.7: [**rpc-threads.c**] Spawning a fresh thread to serve a remote request.

```c
#include "pm2.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 ();
    }
}
```

Program 3.8: [**`rpc-threads1.c`**] Spawning threads to serve requests, an extended example (Follows up on page 33).

```
ravel% pm2load rpc-threads
[Threads : 2 created, 0 imported (0 cached)]

ravel% pm2logs
*** Process 1 on debussy:
[Threads : 3 created, 0 imported (0 cached)]
Hello world!
*** Process 2 on faure:
[Threads : 2 created, 0 imported (0 cached)]
```

To have a bit more fun on closing this part, let us design a slightly more complex program, as the one on Figure 3.8. 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.

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

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

  pm2_init (&argc, argv);

  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 ();
    }

  pm2_exit ();

  return 0;
}
```

Program 3.8 (Cont'd from page 32): [**`rpc-threads1.c`**] Spawning threads to serve requests, an extended example.

```
ravel% pm2load rpc-threads1
[Threads : 3 created, 0 imported (0 cached)]
Sending string: Init
Received back string: Init 1 2

ravel% pm2logs
*** Process 1 on debussy:
[Threads : 3 created, 0 imported (0 cached)]
Passing on string: Init 1
*** Process 2 on faure:
[Threads : 3 created, 0 imported (0 cached)]
Passing on string: Init 1 2
```

It is a good point to clarify the notion of a *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 exactly use one virtual node per physical node, so that the two notions just match together. But nothing in PM2 requires such an identification. PM2 virtual nodes may be located at any physical nodes. The precise association is made by the **pm2conf** command. For instance, saying

```
ravel% pm2conf ravel debussy faure ravel debussy faure
```

is just as right as anything else. Two processes are launched on each of the three machines. From the point of view of PM2 programs, this makes no difference.

```
ravel% pm2conf 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

ravel% pm2load rpc-threads1
[Threads : 3 created, 0 imported (0 cached)]
Sending string: Init
Received back string: Init 1 2 3 4 5
```

You may even start all processes on the same machine!

```
ravel% pm2conf 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% pm2load rpc-threads1
[Threads : 3 created, 0 imported (0 cached)]
Sending string: Init
Received back string: Init 1 2 3 4 5
```

Also, there is no reason why this machine should be the one which you are logged in. Any machine can make it as well!

```
ravel% pm2conf 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% pm2load rpc-threads1
[Threads : 3 created, 0 imported (0 cached)]
Sending string: Init
Received back string: Init 1 2 3 4 5
```

Chapter 4

Customizing and debugging PM2

$Id: customizing.tex,v 1.4 2001/04/18 15:13:12 bouge Exp $

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 explain in detail below. Technically speaking, a PM2 module is a library to be linked with your program. For instance, module **pm2** is library **libpm2.a**. It is the job of the **pm2-config** utility to automagically generate the right compiling and linking set of directives for the various libraries.

```
# 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 one. Such kind of decisions have a strong influence on the resulting PM2 libraries. Changing the network from Myrinet to SCI amounts to recompiling the Madeleine module. Changing the mode from debugging to optimizing actually results in recompiling all the libraries from scratch. This is a not trivial operation, which consumes time and resources.

NB *LB to RN: Is the name of the C compiler included in the flavor?*

These two 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: Marcel, Madeleine, DSM, and various other ancillary modules. 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 following.

- The list of all currently available flavors is provided by command **`pm2-flavor list`**.
- At any point, the currently active flavor is the value of shell variable **`PM2_FLAVOR`**. Alternatively, most PM2 commands honor 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 variable **`PM2_FLAVOR`**.

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

- Flavor **`xxx`** is kept in file **`${PM2_HOME}/.pm2/flavors/xxx`**, as provided by command **`pm2-flavor find --flavor=xxx`**.

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

- The list of options defined by flavor **`xxx`** is provided by command **`pm2-flavor get --flavor=xxx`**.
- The internal consistency of flavor **`xxx`** can be check by command **`pm2-flavor check --flavor=xxx`**. A flavor may be regenerated using command **`pm2-flavor regenerate --flavor=xxx`**.
- When a module is compiled with flavor **`xxx`**, then the resulting object code is placed into directory **`${PM2_BUILD_DIR}/xxx`**. You may thus have several versions of PM2 ready for use at the same time.
- To compile and link a user program with flavor **`xxx`**, use the flags generated by commands **`pm2-config --cflags`** and **`pm2-config --libs`**, as exemplified in the Makefile on Figure 3.1.

4.2 Making your own, personal PM2 flavors

You know *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 flavor **`pm2`** since your (PM2-)birth:

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

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, 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
```

The binary will be installed in **${PM2_HOME}/.pm2/bin/ezflavor_AAA_SSS**, where **AAA** is your architecture and **SSS** is your system. The root script **${PM2_ROOT}/bin** automagically detects these parameters and redirect your call to the right binary file. Therefore, you can use **ezflavor** from different platform without conflict.

The first step is 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

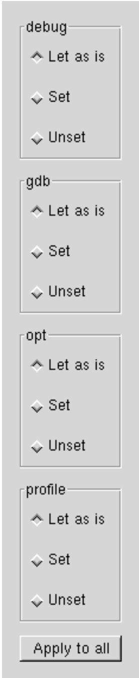

Figure 4.2: The **Common options** window of the **ezfalvor** utility.

option is turned on for this module: **opt**, which makes the module run in optimized mode.

- Select the **mad2** 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 3.2.

- 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. That's right: what you need is a **pm2-debug** flavor, man!

Check in the **debug** option in the **pm2** module. Do it for the application, too, and also for the **mad2** module and the **dsm** module, too. Edit the name of the new flavor from **pm2** to **pm2-debug**, and save it. The new flavor is checked for consistency and saved under into the **${PM2_HOME}/.pm2/flavors/pm2-debug** file. You may check that the file actually contains the modified flavor:

```
ravel% more ${PM2_HOME}/.pm2/flavors/pm2-debug
```

```
# Flavor pm2-debug
[...]
### SETTINGS: --pm2=debug --pm2=opt
[...]
### SETTINGS: --common=debug --common=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=pm2-debug
[...]           # Well, a bunch of lines here...
```

Everything should run smoothly, without any warning...

Once you are done, set the **PM2_FLAVOR** variable to **pm2-debug** and compile your favorite PM2 program using a "dynamic" Makefile as the one on Figure 3.1. Well, ideally, should you not make any difference with the **pm2** run.

Just for fun, modify the **pm2-debug** flavor so as to enable the **isoaddr_trace** in the **pm2** module, for instance. Recompile PM2 with this modified flavor. Recompile your program, for instance the simplistic **hello** on Figure 3.2 and run it. You should see a lot of rather cryptic debugging information:

```
ravel% pm2load hello
Alloc page table(6291456)
index 0: 1
index 1: 1
index 2: 1
[...]
isoaddr_malloc(65536)
isoaddr_malloc: got 16 slots locally
[...]
Hello World!
isoaddr_free: added to stack cache: affe0000 (index = 15)
[Threads : 2 created, 0 imported (0 cached)]
Flushing slot cache...
Flushing stack cache...
Flushing migration cache...
Isoaddr exited
```

> **Warning** *Be careful that the simple Makefile of Figure 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* modules at the same time, together with the application. This holds for the **debug**, **gdb**, **opt** (optimization) and **profile** options.

ezflavor provides such a facility: click on the **View/Common options** menu to open a specialized window. An example is displayed on Figure 4.2.

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**. The default is a line-oriented interface:

```
ravel% pm2-config-flavor --text
================= Flavors management ==================
Existing flavors :
default leonie leoparse mad1 mad2
marcel marcel-act marcel-actsmp marcel-mono marcel-smp
pm2 pm2-mad1 pm2-mad2 pm2-marcel swann

******************************************************
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
```

> **Warning** *In the current version of the system, no backup copy of a flavor is done prior editing. Be careful!*

NB *LB to RN: How should the user could recover the setting of the initial* **pm2** *flavors, in case of accidental corruption?*

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 few tools already exist to assist the programmer in it. 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 is equipped with powerful profiling facilities, to be introduced in Section 6.2. In this section, we rather concentrate on the tools available 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 modules. With the **debug** option enabled, the modules of PM2 are careful to check consistency conditions along the execution, so that abnormal situation may be detected early. You may also wish to concentrate on memory allocations by enabling the **safe_malloc** and the **parano_malloc** options of module **tbx**, the toolbox common to all PM2 modules. If you suspect a problem with synchronization objects, you will find useful options in the **marcel** module, and in the **dsm** module for the inter-node synchronization actions induced by the DSM consistency protocols.

Assuming that you are using the **pm2** flavor, make up a **pm2-debug** one. Set your **PM2_FLAVOR** shell variable to this value, and recompile PM2 with this new flavor.

```
ravel% setenv PM2_FLAVOR pm2-debug
ravel% cd ${PM2_ROOT}
ravel% make
```

Then, recompile the programs of interest with this new flavor.

> **Warning** *Make sure that they are* actually *recompiled: better 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 regular **stdout** one. Also, you should always use the *thread-safe* version of **(f)printf**, called **t(f)printf**. This will prevent multiple Marcel thread to print concurrently, leading to scrambled text. Use the **pm2logs** utility to view the remote logs. It may be useful to force logging also on the main node 0: use **pm2load -l**. Then, **pm2logs** will recover exactly one log per (logical) node.

You may also require "immediate" output from remote node by using the **pm2_(f)printf** routine. Each time it is called, the output is packed into a buffer and sent to main node 0 to be immediately displayed there. The output is prefixed by **[t<i>]**, where **<i>** is the logical number of the originator node. This is quite convenient, but the user has to be aware that the additional messages generated in this way *do interfere* with the original communication scheme of the program under debugging. In particular, they may generate additional deadlocks! Also, these additional messages have to be considered when designing the termination scheme of the program. The **pm2_halt** function should not be called by a node *before* it is guaranteed that all communications, whatever their origin, have been handled by the daemon service threads. Deadlock may otherwise occur.

The **pm2load** utility honors a number of options. The most useful one in a first place is **pm2load -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, **pm2load** 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. Observe also that the windows of nodes other than the main one will only pop up after the main node has started and called its **pm2_main** routine. Then, the node is blocked until you have typed **r** in the window of each subsidiary node. Before starting the nodes, you may add breakpoints and require all kind of funcionalities 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 **^D** in each window will close it and terminate the program.

> **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 grey, 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** is *not* aware of the Marcel threads, so that one cannot rely upon the value of the local variables of the threads in general. Only the values of the global variables are certified. Therefore, you may wish to equip your program with additional debugging code to spy into static, global variables the contents of dynamic, local variables.

NB *LB to RN: Is the paragraph above correct?*

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**
lock_task	**ma_lock_task**	**marcel**
unlock_task	**ma_unlock_task**	**marcel**

Chapter 5

Mastering PM2

$Id: mastering.tex,v 1.4 2001/04/18 15:13:13 bouge Exp $

Congratulation! You are now knowledgeable in PM2, ready to write your own programs. Interested in getting to the *Master* level? Just keep on reading!

5.1 Invoking services synchronously

The problem of gracefully terminating a distributed program is at the very heart of this area of programming. The problem is even sharper with PM2 programs because of the multithreaded behavior which dramatically increases the asynchronism.

PM2 provides the programmer with a simple mechanism to solve the problem, at least in most cases. It is called *completion*. Essentially, a *completion* is a token which is passed along during the computation. The idea is that the creator of the token sends it along to its communication partner. It may eventually block and **`wait`** for receiving back the completion token. When the partner has completed its task, it can **`signal`** it to the token creator by sending back the token to it. Observe that the completion token can travel across many partners before being used.

Consider a typical use of such a mechanism in Program 5.1. It is a minimal variation of Program 3.7. The only difference is that node 0 is now in charge of triggering the termination by calling the **`pm2_halt`** routine. As observed previously, it cannot do it safely in general. It has to make sure beforehand that all the communications have completed. This is precisely the role of the completion token. Node 0 creates token **`c`** with the **`pm2_completion_init`** routine. Observe that only node 0 does it, and that it occurs in the concurrent phase of the program, after the call to **`pm2_init`**. Two extra parameters can be specified on creating a completion token: a specific handler function and additional arguments. Setting the last two parameters to the default **`NULL`** value is sufficient in most usual cases.

Then, the completion token is handled very much like an ordinary object. It can be packed together with other data into a messages using the **`pm2_pack_completion`** routine, with the usual mode flags. It can be similarly unpacked with the **`pm2_unpack_completion`**

```
#include "pm2.h"

#define SIZE 32

static int service_id;
char msg[SIZE];

static void
f (void *arg)
{
  pm2_completion_t c;

  pm2_unpack_byte (SEND_CHEAPER, RECV_CHEAPER, msg, SIZE);
  pm2_unpack_completion (SEND_CHEAPER, RECV_CHEAPER, &c);
  pm2_rawrpc_waitdata ();

  tprintf ("%s\n", msg);
  pm2_completion_signal (&c);
}

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

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

  pm2_init (&argc, argv);

  if (pm2_self () == 0)
    {
      pm2_completion_t c;
      pm2_completion_init (&c, NULL, NULL);

      strcpy (msg, "Hello world!");

      pm2_rawrpc_begin (1, service_id, NULL);
      pm2_pack_byte (SEND_CHEAPER, RECV_CHEAPER, msg, SIZE);
      pm2_pack_completion (SEND_CHEAPER, RECV_CHEAPER, &c);
      pm2_rawrpc_end ();

      pm2_completion_wait (&c);
      pm2_halt ();             /* Correct!!! */
    }

  pm2_exit ();

  return 0;
}
```

Program 5.1: [**`rpc-completion.c`**] Using completion token to garcefully terminate a PM2 program.

routine. Finally, one can send back the token to its creator using the **`pm2_completion_signal`** routine. On its side, the token creator blocks waiting for the token using the **`pm2_completion_wait`** routine. In contrast with Program 3.7, it is now guaranteed that the **`pm2_halt`** function is called on node 0 *after* the thread on node 1 has completed its job!

Any number of tokens can be used within a PM2 program. Also, observe that the **`completion_signal`** involves RPC communications in the background. It is up to the user to guarantee that no deadlock may occur.

NB *LB to RN: Please, proofread and expand the last paragraph. I am not sure about the (probably many) pitfalls here...*

5.2 Thread migration

5.2.1 The basic case: migrating oneself

We now turn to the most amazing feature of PM2: thread migration. The essential idea is quite simple. At some point, a thread can request the local scheduler to migrate some given thread to another node. This *victim* thread may even be the calling thread itself. The victim is dynamically preempted in the scheduler queue. Its set of resources (descriptors, private stack, dynamically allocated memory area) are packed into a buffer (*marshaling*) and a RPC is issued to the destination node with the packed resources as argument. The service associated with this RPC consists in creating a new thread, granting it all the packed resources and restarting it. As far as the victim thread is concerned, this is completely transparent. It is not *aware* of the migration, which merely amounts to an ordinary context switch. Yet, the global environment of the thread is now the one provided by the new node: global variables, node identifier, etc. In particular, thread migration does not make any attempt to keep the system descriptors consistent across the migration: open files, pipes, signals, etc. Mixing migration and I/O has requires uttermost care.

Obviously, some circumstances require a thread to protect itself against migration. This occurs for instance on exchanging messages, doing I/O, etc. PM2 provides the programmer with a **`pm2_enable_migration`** and **`pm2_disable_migration`** routines to manage this aspect. By default, migration is *disabled*.

Program 5.2 is a simple example of thread migration. Each of the nodes of the configuration stores into its global variable **`hostname`** the name of the hosting physical machine. Then, node 0 spawns a thread which declares itself ready for migration. This thread migrates from node to node in a round-robin way for a couple of hops, and eventually stops on a node (which happens not to be the same as its starting node). The **`pm2_halt`** routine is called on the final node.

On each hop, it prints the node number provided by PM2 together with the contents of variable **`hostname`** of its *current global environment*. Observe that this variable is set in the prelude phase, *before* the call to the **`pm2_init`** routine. It would be incorrect to do it later, as no assumption may be made about the time the incoming thread will be unpacked and restarted on the destination node.

```
#include "pm2.h"

char hostname[128];

void
f (void *arg)
{
  int i, proc;

  pm2_enable_migration ();

  proc = pm2_self ();
  for (i = 0; i < (2 * pm2_config_size () + 1); i++)
    {

      tprintf ("Hop %d: I am on node %d, host %s...\n",
               i, pm2_self (), hostname);

      proc = (proc + 1) % pm2_config_size ();
      tprintf ("Hop %d: Leaving to node %d\n", i, proc);

      pm2_migrate_self (proc);
    }
  pm2_halt ();
}

int
pm2_main (int argc, char *argv[])
{
  gethostname (hostname, 128);

  pm2_init (&argc, argv);

  if (pm2_self () == 0)
    {                           /* Master process */
      pm2_thread_create (f, NULL);
    }

  pm2_exit ();
  return 0;
}
```

Program 5.2: [**migration.c**] Migrating a thread across the configuration.

```
ravel% pm2conf ravel debussy faure
The current PM2 configuration contains 3 host(s) :
0 : ravel
1 : debussy
2 : faure

ravel% pm2load migration
[Threads : 5 created, 2 imported (0 cached)]
Hop 0: I am on node 0, host ravel...
Hop 0: Leaving to node 1
Hop 3: I am on node 0, host ravel...
Hop 3: Leaving to node 1
Hop 6: I am on node 0, host ravel...
Hop 6: Leaving to node 1

ravel% pm2logs
*** Process 1 on debussy:
[Threads : 5 created, 3 imported (0 cached)]
Hop 1: I am on node 1, host debussy...
Hop 1: Leaving to node 2
Hop 4: I am on node 1, host debussy...
Hop 4: Leaving to node 2
*** Process 2 on faure:
[Threads : 4 created, 2 imported (0 cached)]
Hop 2: I am on node 2, host faure...
Hop 2: Leaving to node 0
Hop 5: I am on node 2, host faure...
Hop 5: Leaving to node 0
```

As usual, there is no restriction whatsoever on the association between the physical machine and the logical nodes of PM2. For instance, you could try to run the program on the same physical node. It works just as well!

```
ravel% pm2conf ravel ravel ravel
The current PM2 configuration contains 3 host(s) :
0 : ravel
1 : ravel
2 : ravel

ravel% pm2load migration
[Threads : 5 created, 2 imported (0 cached)]
Hop 0: I am on node 0, host ravel...
Hop 0: Leaving to node 1
Hop 3: I am on node 0, host ravel...
Hop 3: Leaving to node 1
Hop 6: I am on node 0, host ravel...
Hop 6: Leaving to node 1
```

```
ravel% pm2logs
*** Process 1 on ravel:
[Threads : 5 created, 3 imported (0 cached)]
Hop 1: I am on node 1, host ravel...
Hop 1: Leaving to node 2
Hop 4: I am on node 1, host ravel...
Hop 4: Leaving to node 2
*** Process 2 on ravel:
[Threads : 4 created, 2 imported (0 cached)]
Hop 2: I am on node 2, host ravel...
Hop 2: Leaving to node 0
Hop 5: I am on node 2, host ravel...
Hop 5: Leaving to node 0
```

5.2.2 Combing migration and completion

The main problem in the previous program is to *gracefully* terminate the computation. The easy solution implemented in Program 5.2 was to let the moving thread initiate the termination. However, this is obviously a rather specific trick, which does not generalize!

The idea is to combine the migration facility together with the completion one. On spawning a migrating thread, the spawning thread generates a completion token and passes it on to the migrating thread. Then, it blocks, waiting on the completion to be signaled. When the thread has completed its series of hops, it signals the completion to its originator, which can then initiates the termination phase. Observe that this crucially relies on the fact that a completion includes enough information to be routed back to its originator, and that this information is physically stored *explicitly* in the completion (in opposite to being accessed through pointers). If the completion token is stored within the resources of the moving thread, for instance, its stack as a local variable, then it gets migrated together with the thread and can be used from any node (including the originator node!)

This approach is exemplified in Program 5.3. Observe how the completion is passed from its originator to the moving thread. The completion is created in a global variable on node 0. The first action of the created thread is to copy the completion into a local variable, so that is gets migrated together with the rest of the resources. This uses a special routine called **`pm2_completion_copy`**.

> **Warning** *You are strongly advised not to copy completion directly as in* **`new_c = old_c`**, *though it works in the current version of PM2. Instead, you should say*
>
> `pm2_completion_copy(&new_c, &old_c)`
>
> *to guarantee that a* physical *copy of the relevant information is properly stored into the completion. Unfortunately for the oblivious user, PM2 does very little verification about the proper use of such auxiliary routines. It also*

```
#include "pm2.h"

char hostname[128];
pm2_completion_t c;

void
f (void *arg)
{
  int i, proc;
  pm2_completion_t my_c;

  pm2_completion_copy (&my_c, &c);

  pm2_enable_migration ();

  proc = pm2_self ();
  for (i = 0; i < (2 * pm2_config_size () + 1); i++)
    {

      tprintf ("Hop %d: I am on node %d, host %s...\n",
               i, pm2_self (), hostname);

      proc = (proc + 1) % pm2_config_size ();
      tprintf ("Hop %d: Leaving to node %d\n", i, proc);

      pm2_migrate_self (proc);
    }
  pm2_completion_signal (&my_c);
}


int
pm2_main (int argc, char *argv[])
{
  gethostname (hostname, 128);

  pm2_init (&argc, argv);

  if (pm2_self () == 0)
    {                           /* Master process */
      pm2_completion_init (&c, NULL, NULL);

      pm2_thread_create (f, NULL);

      pm2_completion_wait (&c);
      pm2_halt ();
    }

  pm2_exit ();
  return 0;
}
```

Program 5.3: [**migration1.c**] Using completions in presence of migration.

provide few facilities to debug them. Just believe me: You'd better write a correct code at the first attempt!

The technique used above, having the originator thread passing a completion token to the moving thread, is rather contrived. A better approach is exemplified in Program 5.4. A pointer to the completion is passed as an argument to the thread creation function. The spawned thread allocate storage locally for a completion, and copies the argument completion into its personal resources using the **`pm2_completion_copy`** routine:

```
pm2_completion_copy(&my_c, (pm2_completion_t *)arg)
```

This saves the allocation of some global storage and it is much cleaner. Yet, observe that this works because the storage for the original completion is guaranteed to remain accessible at the time of the copy. There is a subtle synchronization game here, as it will be only destroyed *after* the completion is signaled back to its originator. In case of doubt, it would have been much safer to allocate the original completion in the global space, as in Program 5.2.

If the moving thread is started on a *remote* node using a RPC, then the completion can be passed as an argument to the RPC call, as demonstrated in Program 5.5. This is probably the cleanest way of all. Unfortunately, the semantics of PM2 does not allow a RPC to the local node, which rules out this technique in the case of a local creation.

5.2.3 Migrating other threads

In the Section 5.2.1, we have considered the simple case: a thread migrates itself off its current node. The migration facility of PM2 is much more powerful, as it allows a thread to migrate off other sibling threads, provided these threads have agreed on getting migrated.

Consider the program displayed on Figure 5.6. It is a good opportunity to demonstrate advanced techniques for threads management. The idea is the following: Node 0 spawns 4 children threads, numbered from 0 to 3. (Remind that threads disregard migration by default.) Then, the children threads with an even number agree on migrating, and all children threads loop, repeatedly yielding control to the scheduler with the **`marcel_yield`** routine until variable **`barrier`** takes a non null value. When such a value has been recorded, the children threads issue a message and terminate. On its side, the master thread on node 0 records the list of **`MIGRATABLE`** threads, and requests the migration of all of them to node 1. Then, it issues a RPC to node 1 to set variable **`barrier`** to 1, and does so for its own node. Finally, the master thread calls the **`halt`** routine to stop all daemon service threads.

```
ravel% pm2load migrate-group
Initialization completed on node 0
Thread 0: Created on node 0, host ravel
Thread 1: Created on node 0, host ravel
Thread 2: Created on node 0, host ravel
Thread 3: Created on node 0, host ravel
2 threads among 4 migrated off to node 1
```

```
#include "pm2.h"

char hostname[128];


void
f (void *arg)
{
  int i, proc;
  pm2_completion_t my_c;
  pm2_completion_copy (&my_c, (pm2_completion_t *) arg);

  pm2_enable_migration ();

  proc = pm2_self ();
  for (i = 0; i < (2 * pm2_config_size () + 1); i++)
    {

      tprintf ("Hop %d: I am on node %d, host %s...\n",
               i, pm2_self (), hostname);

      proc = (proc + 1) % pm2_config_size ();
      tprintf ("Hop %d: Leaving to node %d\n", i, proc);

      pm2_migrate_self (proc);
    }
  pm2_completion_signal (&my_c);
}


int
pm2_main (int argc, char *argv[])
{
  gethostname (hostname, 128);

  pm2_init (&argc, argv);

  if (pm2_self () == 0)
    {                           /* Master process */
      pm2_completion_t c;
      pm2_completion_init (&c, NULL, NULL);

      pm2_thread_create (f, (void *) (&c));

      pm2_completion_wait (&c);
      pm2_halt ();
    }

  pm2_exit ();
  return 0;
}
```

Program 5.4: [**migration2.c**] Passing completions as parameters.

```
#include "pm2.h"

char hostname[128];

static int service_id;

void
f (void *arg)
{
  int i, proc;
  pm2_completion_t my_c;

  pm2_unpack_completion (SEND_CHEAPER, RECV_CHEAPER, &my_c);
  pm2_rawrpc_waitdata ();

  pm2_enable_migration ();

  proc = pm2_self ();
  for (i = 0; i < (2 * pm2_config_size () + 1); i++)
    {

      tprintf ("Hop %d: I am on node %d, host %s...\n",
               i, pm2_self (), hostname);

      proc = (proc + 1) % pm2_config_size ();
      tprintf ("Hop %d: Leaving to node %d\n", i, proc);

      pm2_migrate_self (proc);
    }
  pm2_completion_signal (&my_c);
}

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

Program 5.5: [**migration3.c**] Passing completions through a RPC call (Follows up on page 54).

```
int
pm2_main (int argc, char *argv[])
{
  gethostname (hostname, 128);
  pm2_rawrpc_register (&service_id, service);

  pm2_init (&argc, argv);

  if (pm2_self () == 0)
    {                           /* Master process */
      pm2_completion_t c;
      pm2_completion_init (&c, NULL, NULL);

      pm2_rawrpc_begin (1, service_id, NULL);
      pm2_pack_completion (SEND_CHEAPER, RECV_CHEAPER, &c);
      pm2_rawrpc_end ();

      pm2_completion_wait (&c);
      pm2_halt ();
    }

  pm2_exit ();
  return 0;
}
```

Program 5.5 (Cont'd from page 53): [**`migration3.c`**] Passing completions through a RPC call.

```
Issuing RPC to node 1
Just halting
Thread 3: Now on node 0, host ravel
Thread 1: Now on node 0, host ravel
[Threads : 6 created, 0 imported (0 cached)]

ravel% pm2logs

*** Host debussy, process 1:
[Threads : 4 created, 2 imported (0 cached)]
Initialization completed on node 1
Service activated on node 1, host debussy
Thread 0: Now on node 1, host debussy
Thread 2: Now on node 1, host debussy

*** Host faure, process 2:
[Threads : 2 created, 0 imported (0 cached)]
Initialization completed on node 2
```

Let's now look more closely at the program. First, observe how the master thread spawns the children threads and synchronize with them using a completion token. Each child thread is created with two parameters: a completion token **`c`** and a number **`i`**. As the thread creation

```c
#include "pm2.h"

static int service_id;
char hostname[128];

#define N 4
marcel_t threads[N];
int where = 1;

struct arg
{
  pm2_completion_t *cp;
  int i;
};

volatile int barrier = 0;

void
f (void *arg)
{
  struct arg *argp;
  pm2_completion_t my_c;
  int i;

  argp = (struct arg *) arg;
  pm2_completion_copy (&my_c, argp->cp);
  i = argp->i;

  if (i % 2 == 0)
    pm2_enable_migration ();

  tprintf ("Thread %d: Created on node %d, host %s\n",
           i, pm2_self (), hostname);
  pm2_completion_signal (&my_c);

  for (;;)
    {
      int tmp;
      tmp = barrier;
      if (tmp != 0)
        break;
      marcel_yield ();
    }

  tprintf ("Thread %d: Now on node %d, host %s\n",
           i, pm2_self (), hostname);
}
```

Program 5.6: [**migrate-group.c**] Migrating off a group of threads (Follows up on page 56).

```
static void
service (void)
{
  pm2_rawrpc_waitdata ();
  tprintf ("Service activated on node %d, host %s\n",
           pm2_self (), hostname);
  barrier = 1;
}

int
pm2_main (int argc, char *argv[])
{
  int i;
  int n;

  gethostname (hostname, 128);
  pm2_rawrpc_register (&service_id, service);

  pm2_init (&argc, argv);

  tprintf ("Initialization completed on node %d\n", pm2_self ());

  if (pm2_self () == 0)
    {                            /* Master process */
      for (i = 0; i < N; i++)
        {
          pm2_completion_t c;
          struct arg arg;
          pm2_completion_init (&c, NULL, NULL);
          arg.cp = &c;
          arg.i = i;
          pm2_thread_create (f, (void *) (&arg));
          pm2_completion_wait (&c);
        }

      pm2_freeze ();
      pm2_threads_list (N, threads, &n, MIGRATABLE_ONLY);
      pm2_migrate_group (threads, n, where);
      tprintf ("%d threads among %d migrated off"
               "to node %d\n", n, N, where);

      tprintf ("Issuing RPC to node %d\n", where);
      pm2_rawrpc_begin (where, service_id, NULL);
      pm2_rawrpc_end ();
      barrier = 1;

      tprintf ("Just halting\n");
      pm2_halt ();
    }
  pm2_exit ();
  return 0;
}
```

Program 5.6 (Cont'd from page 55): [**`migrate-group.c`**] Migrating off a group of threads.

routine only accepts one argument of type **`void *`**, an auxiliary structure has to be used: the two parameters have to be packed into a structure; a pointer to the structure is passed as the argument of the **`pm2_thread_create`** routine; the structure is eventually unpacked by the thread.

> **Warning** *This scheme is correct* only *because of the synchronization implemented by the completion token: the master thread cannot spawn another thread before the current thread has unpacked the structure. Once one the completion token has been received from each children thread, then the master thread is guaranteed that all of them have been created, and have decided upon whether they enable their own migration or no.*

Consider now how the master thread can migrate the children threads. The first step is to store the children threads to be migrated into an array. This can be done by hand, but PM2 provides a specialized routine to do the work. A call **`pm2_threads_list(N, threads, &n, MIGRATABLE_ONLY)`** takes as parameters an array of threads **`threads[N]`** (a PM2 thread has type **`marcel_t`**) and a selection condition. A number of condition are available, which can be or-ed and and-ed as usual for flags. Here, we use the **`MIGRATABLE_ONLY`** flag, which does exactly what you guess. Well, not exactly, as the flag is (silently) and-ed with the **`NOT_BLOCKED_ONLY`** flag: only the threads which have enabled migration *and* are not currently blocked are stored into the array. At most **`N`** such threads are stored, and the number of actually stored threads is recorded into **`n`**. Observe that the calling thread is listed, too, if it meets the condition. This is not the case in this example.

You may object that this sampling operation may yield inconsistent result as the sampled threads are running asynchronously with the sampling one, very much as in the **`ps`** command of Unix. This is not the case, as the **`pm2_threads_list`** routine internally disables Marcel, the thread scheduler of PM2 using the **`lock_task`**/**`unlock_task`** internal routines of Marcel. However, the user has no control about the exact sampling time, but through the explicit synchronization which have been introduced into the program. Once the list of *victim* threads to be migrated has been constructed, then PM2 can migrate all of them to a common node using the **`pm2_migrate_group`** routine. This node may well be the local node. Of course, attempting to migrate a thread which has disabled migration to a remote node raises an error.

This raises a new problem: how can the user make sure that the threads sampled out with the **`pm2_threads_list`** routine are still enabling migration? Or even, that they are still running on the local node? Or even, that they are still alive? Well, the only general solution is to guarantee the atomicity of the two successive calls. This is the role of the **`pm2_freeze`** routine: it disables the scheduler, so that the state of the other threads may not change until the **`pm2_unfreeze`** routine is called. Of course, this also suspends all daemon service threads, so that no message can be handled. This is up to the user to guarantee that freezing the scheduling will not cause any deadlock in the system, for instance by overfilling communication buffers. Also, calls to **`pm2_freeze`**/**`pm2_unfreeze`** may be nested. It is the responsibility of the user to ensure proper nesting.

Well, well, well... Imagine the following scenario. Consider a thread which enables *its own migration*; it freezes the scheduling on the *local* node, samples the migratable threads, and calls the **pm2_migrate_group** routine to migrate them away to some *remote* node. It is itself migrated! Calling the **pm2_unfreeze** routine afterwards will most probably raise an error, as no call to **pm2_freeze** has ever been made on the *remote* node. Also, nobody is left on the *local* node to unfreeze the scheduler, and this would most probably result in a deadlock. Interesting problem, isn't it? The only solution is that PM2 calls the **pm2_freeze** routine on the local node *on behalf* of the migrated thread. This somewhat non standard call is actually included at the end of the **pm2_migrate_group** routine. This is why there is no *explicit* call to **pm2_unfreeze** in our program. It is in fact hidden within the call to **pm2_migrate_group**.

Observe that the program on Figure 5.6 uses a variable **barrier** which is asynchronously read by many threads and written by one thread. Strictly speaking, it is not correct to do so, as there is no reason why the writing operation should be atomic. Actually, the readers could in general read various intermediate values from the variable, depending for instance on the order in which the bits are written. One should guard all accesses to a shared variable with a *mutual exclusion object* (a mutex, for short). Mutexes are described in Section 6.1.2. However, the behavior of this very program does not depend on concurrent reading and writing, as the variable is only asynchronously tested against 0. Spotting a bit set to 1 early or late does not matter: for each thread, once a non-null value has been read, then no null value will ever be read afterwards. Such a situation is well-known in the area of distributed algorithm: it is called a *stable condition*.

NB *LB to RN: Difficult section. Please, proofread!*

5.2.4 Iso-allocation and dynamically allocated data

On migrating, a thread is moved together with its private resources: its private descriptors and its private stack. All these resources are allocated by PM2 so that the virtual addresses remains invariant upon migration. This allows to freely use pointers in the stack of a migrating thread. This is exemplified in Program 5.7. Variable **x** is allocated in the private stack of the thread on node 1. After migration to node 2, the value of **x** *and its address* are left unchanged.

```
ravel% pm2logs
*** Process 1 on debussy:
[Threads : 3 created, 0 imported (0 cached)]
First, I am on node 1, host debussy... &x = affafda0, x = 1234
*** Process 2 on faure:
[Threads : 3 created, 1 imported (0 cached)]
Now, I am on node 2, host faure... &x = affafda0, x = 1234
```

This amazing feature is made possible thanks to the *iso-allocation* technique used by PM2 to allocate memory dynamically. The whole virtual space of the nodes is managed in a consistent way. If some storage is granted to some thread at one node, then the iso-allocation technique guarantees that the corresponding range of virtual addresses will not be used by any other node in the configuration. Then, on a migration, the storage area can just get

```
#include "pm2.h"

char hostname[128];
static int service_id;

void
f (void *arg)
{
  int proc;

  int x = 1234;

  pm2_rawrpc_waitdata ();

  pm2_enable_migration ();
  proc = pm2_self ();

  tprintf
    ("First, I am on node %d, host %s... &x = %p, x = %d\n",
     pm2_self (), hostname, &x, x);

  proc = (proc + 1) % pm2_config_size ();
  pm2_migrate_self (proc);

  tprintf
    ("Now, I am on node %d, host %s... &x = %p, x = %d\n",
     pm2_self (), hostname, &x, x);

  pm2_halt ();
}

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

int
pm2_main (int argc, char *argv[])
{
  gethostname (hostname, 128);
  pm2_rawrpc_register (&service_id, service);
  pm2_init (&argc, argv);

  if (pm2_self () == 0)
    {                           /* Master process */
      pm2_rawrpc_begin (1, service_id, NULL);
      pm2_rawrpc_end ();
    }
  pm2_exit ();
  return 0;
}
```

Program 5.7: [**iso-alloc.c**] Iso-allocation enables unlimited use of pointers in migrating threads.

copied exactly at the same virtual address in the destination node. All pointers remain valid without any dependency on the compiler nor any optimization strategy! You can therefore freely manipulate linked lists within migrating threads without the smallest worrying...

All storage requests in PM2 are handled with this technique, including the storage allocated for the private stack on creating a fresh thread. However, it is also available for explicit, dynamic storage allocation by the programmer, as a substitute for the **`malloc`**/**`free`** functions. They are just called **`pm2_isomalloc`** and **`pm2_isofree`**, with the same profiles as the usual **`malloc`** and **`free`** functions. Dynamic memory storage allocated with these functions are included in the private resources of the threads and are migrated together with the thread. Because of the iso-allocation technique, the storage is mapped at the *same* virtual address in all nodes, so that pointers can be freely used: you can use them within a single storage area, but also across areas, and even between the private stack and the storage areas.

NB *LB to GA: Please, review this carefully!*

In Program 5.8, the thread *iso-allocates* two storage area at addresses **`p`** and **`q`**. The variable located at **`p`**, (that is, **`*p`**) holds the address **`q`**. The variable located a **`q`** (that is, **`*q`**) holds some data. Thus, **`**p`** returns this data through a double indirection. On allocating the storage areas, the **`pm2_isomalloc`** function registers them in the private descriptors of the thread. When the thread is about migrating, the registered storage areas are packed together with its descriptors and its stack. They are unpacked on the destination node, and installed at the same virtual address as in the origin node. This is why **`**p`** returns the same value *after* the migration, as before.

```
ravel% pm2load isomalloc
First, I am on node 0, host ravel...
p = aff8f040, *p = aff8f080, **p = 1234
[Threads : 3 created, 0 imported (0 cached)]

ravel% pm2logs
*** Host debussy, process 1:
[Threads : 3 created, 1 imported (0 cached)]
Then, I am on node 1, host debussy...
p = aff8f040, *p = aff8f080, **p = 1234
```

A thread may allocate as many such dynamic storage areas as needed. Observe that this allocation primitive is compatible with the usual **`malloc`** primitives. A thread may well allocate storage with both functions within the same program. However, only areas allocated with **`pm2_isomalloc`** shall migrate together with the thread. The result of accessing an address allocated with the ordinary **`malloc`** function, after a migration is of course undefined as demonstrated by Program 5.9: the value of **`*p`** is *not* preserved through the migration!

```
ravel% pm2load isomalloc1
First, I am on node 0, host ravel...
p = 80d2500, *p = 1234
```

```c
#include "pm2.h"

char hostname[128];

void
service (void *arg)
{
  int **p, *q;
  int proc;

  p = (int **) pm2_isomalloc (sizeof (*p));
  q = (int *) pm2_isomalloc (sizeof (*q));

  *p = q;
  *q = 1234;

  pm2_enable_migration ();
  proc = pm2_self ();

  tprintf ("First, I am on node %d, host %s...\n"
           "p = %p, *p = %p, **p = %d\n",
           pm2_self (), hostname, p, *p, **p);

  proc = (proc + 1) % pm2_config_size ();
  pm2_migrate_self (proc);

  tprintf ("Then, I am on node %d, host %s...\n"
           "p = %p, *p = %p, **p = %d\n",
           pm2_self (), hostname, p, *p, **p);

  pm2_isofree (p);
  pm2_isofree (q);

  pm2_halt ();
}


int
pm2_main (int argc, char *argv[])
{
  gethostname (hostname, 128);
  pm2_init (&argc, argv);
  if (pm2_self () == 0)
    {                            /* Master process */
      pm2_thread_create (service, NULL);
    }
  pm2_exit ();
  return 0;
}
```

Program 5.8: [**`isomalloc.c`**] Allocating dynamic storage which migrates together with the thread.

```
#include "pm2.h"

char hostname[128];

void
f (void *arg)
{
  int *p;
  int proc;

  p = (int *) malloc (sizeof (*p));

  *p = 1234;

  pm2_enable_migration ();
  proc = pm2_self ();

  tprintf ("First, I am on node %d, host %s...\n"
           "p = %p, *p = %d\n", pm2_self (), hostname, p, *p);

  proc = (proc + 1) % pm2_config_size ();
  pm2_migrate_self (proc);

  tprintf ("Then, I am on node %d, host %s...\n"
           "p = %p, *p = %d\n", pm2_self (), hostname, p, *p);

  free (p);

  pm2_halt ();
}


int
pm2_main (int argc, char *argv[])
{
  gethostname (hostname, 128);

  pm2_init (&argc, argv);

  if (pm2_self () == 0)
    {                            /* Master process */
      pm2_thread_create (f, NULL);
    }

  pm2_exit ();
  return 0;
}
```

Program 5.9: [**isomalloc1.c**] Ordinary allocation results in undefined behavior in the presence of migration.

```
[Threads : 3 created, 0 imported (0 cached)]

ravel% pm2logs
*** Host debussy, process 1:
[Threads : 3 created, 1 imported (0 cached)]
Then, I am on node 1, host debussy...
p = 80d2500, *p = 8
```

5.3 Sharing data across nodes: Distributed Shared Memory

Up to now, we have only considered regular threads, which share data with their sibling threads within the same processing node, that is, their hosting Unix process. In regular PM2 programming, no action is made to manage such data and to preserve consistency on migration. For this purpose, PM2 includes a special module called DSM-PM2 (DSM stands for *Distributed Shared Memory*) to provide such a functionality. Variables assigned within *shared storage area* can be accessed from any thread, whatever its hosting node. The system guarantees the consistency of the values observed by the threads according a given semantic model, to be specified by the user through an initialization routine.

Consider Program 5.10. Variable **`shvar`** is declared to be *globally shared* by all the threads of the program. Each processing node but the master node, successively spawns a thread to increment this single shared variable for 20 units, recording the initial and the final value it observed. Finally, the master node prints the value of the variable. As expected, the final value of the variable is exactly as it would have been with a sequence of increments at the master node.

```
ravel% pm2load dsm
shvar=40
[Threads : 6 created, 0 imported (0 cached)]

ravel% pm2logs
*** Host debussy, process 1:
Thread 1 from node 1 finished on node 1: from 0 to 20!
[Threads : 8 created, 0 imported (0 cached)]
Thread 1 on node 1

*** Host faure, process 2:
Thread 2 from node 2 finished on node 2: from 20 to 40!
[Threads : 6 created, 0 imported (0 cached)]
Thread 2 on node 2
```

DSM-PM2 implements a complex protocol to guarantee that the final result is the right one. In this example, the variable is physically installed on the master node: it is called its *home node*. Each time a thread attempts to *read* the value of the variable, it gets a (read-only) copy of the variable from the master node, so that repeated reads are very cheap. Each time a thread attempts to*write* a new value into the variable, all the subsidiary copies of the

```
#include "pm2.h"

#define NB_NODES (pm2_config_size())
#define NB_ITERATIONS 20

static int service_id;

BEGIN_DSM_DATA;
int shvar = 0;
END_DSM_DATA;

void
f (void *arg)
{
  int i;
  int my_name;
  int my_node = pm2_self ();
  pm2_completion_t my_c;
  int initial, final;

  pm2_unpack_int (SEND_CHEAPER, RECV_CHEAPER, &my_name, 1);
  pm2_unpack_completion (SEND_CHEAPER, RECV_CHEAPER, &my_c);
  pm2_rawrpc_waitdata ();

  tprintf ("Thread %d started on node %d\n", my_name, my_node);

  initial = shvar;
  for (i = 0; i < NB_ITERATIONS; i++)
    shvar++;
  final = shvar;

  tprintf ("Thread %d from node %d finished on node %d: "
           "from %d to %d!\n",
           my_name, my_node, pm2_self (), initial, final);
  pm2_completion_signal (&my_c);
}


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

Program 5.10: [**`dsm.c`**] Distributed shared memory in PM2 (Follows up on page 65).

```
int
pm2_main (int argc, char **argv)
{
  int i;
  int thread_id = 0;
  pm2_completion_t c;

  pm2_rawrpc_register (&service_id, service);
  pm2_completion_init (&c, NULL, NULL);

  dsm_set_default_protocol (LI_HUDAK);

  pm2_init (&argc, argv);

  if (pm2_self () == 0)
    {                           /* Master process */
      for (i = 1; i < NB_NODES; i++)
        {
          thread_id++;
          pm2_rawrpc_begin (i, service_id, NULL);
          pm2_pack_int (SEND_CHEAPER, RECV_CHEAPER,
                        &thread_id, 1);
          pm2_pack_completion (SEND_CHEAPER, RECV_CHEAPER, &c);
          pm2_rawrpc_end ();

          pm2_completion_wait (&c);
        }
      tprintf ("shvar=%d\n", shvar);
      pm2_halt ();
    }
  pm2_exit ();
  return 0;
}
```

Program 5.10 (Cont'd from page 64): [**dsm.c**] Distributed shared memory in PM2.

variable are invalidated, and the new value is stored into the master copy of the variable at its home node. This protocol is known in the literature as the Li-Hudak's protocol. It implements a *sequential consistency*: everything happens as if all the read/write actions had been applied *sequentially* to the variable at its home node, though the order in which the operations have been applied is unknown. This is why the program says in the initialization phase: **dsm_set_default_protocol(LI_HUDAK)**.

Other protocols are available in DSM-PM2. For instance, using **MIGRATE_THREAD** instead of **LI_HUDAK** implements another protocol to guarantee sequential consistency, based on dynamic thread migration: each time a thread attempts to access a globally shared variable, to read it as well as to write it, then it silently migrates to the home node of the variable. As you can observe, the result is the same as above, but all the incrementing threads end up on the home node of the shared variable.

```
ravel% pm2load dsm1
Thread 1 from node 1 finished on node 0: from 0 to 20!
Thread 2 from node 2 finished on node 0: from 20 to 40!
shvar=40
[Threads : 4 created, 2 imported (0 cached)]

ravel% pm2logs
*** Host debussy, process 1:
[Threads : 3 created, 0 imported (0 cached)]
Thread 1 on node 1

*** Host faure, process 2:
[Threads : 3 created, 0 imported (0 cached)]
Thread 2 on node 2
```

In the two examples above, each thread accessing shared variable **shvar** is launched in turn, waiting for the previous one to signal completion. Then, no access conflict on the shared variable is to be threatened. What if the thread attempt to increment the variable *concurrently* as in Program 5.12? Actually, the incrementing instruction **shvar++** is compiled into an *sequence* of elementary, atomic machine instructions, something like:

```
{int tmp; tmp = shvar; tmp++; shvar = tmp;}
```

Thus, we have to add extra synchronization objects to guarantee that this sequence of atomic instructions is actually executed in mutual exclusion, as if it was a single instruction. DSM-PM2 provides *global* mutexes for this purpose, as shown in the program. Observe that this mutexes are much more powerful than the regular PM2 mutexes, which are local to a PM2 node.

```
ravel% ./run dsm2
Compiling dsm2.c... for flavor 'pm2'
ravel% pm2conf ravel debussy faure
The current PM2 configuration contains 3 host(s) :
```

```
#include "pm2.h"

#define NB_NODES (pm2_config_size())
#define NB_ITERATIONS 20

static int service_id;

BEGIN_DSM_DATA;
int shvar = 0;
END_DSM_DATA;

void
f (void *arg)
{
  int i;
  int my_name;
  int my_node = pm2_self ();
  pm2_completion_t my_c;
  int initial, final;

  pm2_unpack_int (SEND_CHEAPER, RECV_CHEAPER, &my_name, 1);
  pm2_unpack_completion (SEND_CHEAPER, RECV_CHEAPER, &my_c);
  pm2_rawrpc_waitdata ();

  tprintf ("Thread %d started on node %d\n", my_name, my_node);

  initial = shvar;
  for (i = 0; i < NB_ITERATIONS; i++)
    shvar++;
  final = shvar;

  tprintf ("Thread %d from node %d finished on node %d: "
           "from %d to %d!\n",
           my_name, my_node, pm2_self (), initial, final);
  pm2_completion_signal (&my_c);
}


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

Program 5.11: [**dsm1.c**] Distributed shared memory in PM2, using the **MIGRATE_THREAD** sequential consistency protocol (Follows up on page 68).

```
int
pm2_main (int argc, char **argv)
{
  int i;
  int thread_id = 0;
  pm2_completion_t c;

  pm2_rawrpc_register (&service_id, service);
  pm2_completion_init (&c, NULL, NULL);

  dsm_set_default_protocol (MIGRATE_THREAD);

  pm2_init (&argc, argv);

  if (pm2_self () == 0)
    {                            /* Master process */
      for (i = 1; i < NB_NODES; i++)
        {
          thread_id++;
          pm2_rawrpc_begin (i, service_id, NULL);
          pm2_pack_int (SEND_CHEAPER, RECV_CHEAPER,
                        &thread_id, 1);
          pm2_pack_completion (SEND_CHEAPER, RECV_CHEAPER, &c);
          pm2_rawrpc_end ();

          pm2_completion_wait (&c);
        }
      tprintf ("shvar=%d\n", shvar);
      pm2_halt ();
    }
  pm2_exit ();
  return 0;
}
```

Program 5.11 (Cont'd from page 67): [**dsm1.c**] Distributed shared memory in PM2, using the **MIGRATE_THREAD** sequential consistency protocol.

```
#include "pm2.h"

#define NB_NODES (pm2_config_size())
#define NB_ITERATIONS 20

static int service_id;

dsm_mutex_t mutex;

BEGIN_DSM_DATA;
int shvar = 0;
END_DSM_DATA;

void
f (void *arg)
{
  int i;
  int my_name;
  int my_node = pm2_self ();
  pm2_completion_t my_c;
  int initial, final;

  pm2_unpack_int (SEND_CHEAPER, RECV_CHEAPER, &my_name, 1);
  pm2_unpack_completion (SEND_CHEAPER, RECV_CHEAPER, &my_c);
  pm2_rawrpc_waitdata ();

  tprintf ("Thread %d started on node %d\n", my_name, my_node);

  dsm_mutex_lock (&mutex);
  initial = shvar;
  for (i = 0; i < NB_ITERATIONS; i++)
    shvar++;
  final = shvar;
  dsm_mutex_unlock (&mutex);

  tprintf ("Thread %d from node %d finished on node %d: "
           "from %d to %d!\n",
           my_name, my_node, pm2_self (), initial, final);
  pm2_completion_signal (&my_c);
}


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

Program 5.12: [**dsm2.c**] Distributed shared memory in PM2, with concurrent access to shared variables (Follows up on page 70).

```
int
pm2_main (int argc, char **argv)
{
  int i;
  int thread_id = 0;
  pm2_completion_t c;

  pm2_rawrpc_register (&service_id, service);
  pm2_completion_init (&c, NULL, NULL);

  dsm_set_default_protocol (LI_HUDAK);
  dsm_mutex_init (&mutex, NULL);

  pm2_init (&argc, argv);

  if (pm2_self () == 0)
    {                           /* Master process */
      for (i = 1; i < NB_NODES; i++)
        {
          thread_id++;
          pm2_rawrpc_begin (i, service_id, NULL);
          pm2_pack_int (SEND_CHEAPER, RECV_CHEAPER,
                        &thread_id, 1);
          pm2_pack_completion (SEND_CHEAPER, RECV_CHEAPER, &c);
          pm2_rawrpc_end ();
        }

      for (i = 1; i < NB_NODES; i++)
        pm2_completion_wait (&c);

      tprintf ("shvar=%d\n", shvar);
      pm2_halt ();
    }
  pm2_exit ();
  return 0;
}
```

Program 5.12 (Cont'd from page 69): [**`dsm2.c`**] Distributed shared memory in PM2, with concurrent access to shared variables.

```
0 : ravel
1 : debussy
2 : faure

ravel% pm2load dsm2
shvar=40
[Threads : 10 created, 0 imported (0 cached)]

ravel% pm2logs
*** Host debussy, process 1:
[Threads : 8 created, 0 imported (0 cached)]
Thread 1 started on node 1
Thread 1 from node 1 finished on node 1: from 0 to 20!

*** Host faure, process 2:
[Threads : 6 created, 0 imported (0 cached)]
Thread 2 started on node 2
Thread 2 from node 2 finished on node 2: from 20 to 40!
```

Just for fun, you may wish to try out this last program *without* the mutexes! This is legal in DSM-PM2, and the sequential consistency semantics model for the DSM fully addresses it. However, you should remind that the model specifies the behavior at the level of elementary atomic instructions: reading and writing shared variables, so that the actual program which is run is the one with the expanded form is **shvar++**. I run the experiment on my machine: here is the result:

```
ravel% more dsm3.c
[...]
  // dsm_mutex_lock (&L);
  initial = shvar;
  for (i = 0; i < NB_ITERATIONS; i++)
    shvar++;
  final = shvar;
  // dsm_mutex_unlock (&L);
[...]

ravel% pm2load dsm3
shvar=20
[Threads : 7 created, 0 imported (0 cached)]

ravel% pm2logs
*** Host debussy, process 1:
[Threads : 7 created, 0 imported (0 cached)]
Thread 1 started on node 1
Thread 1 from node 1 finished on node 1: from 0 to 20!

*** Host faure, process 2:
```

```
[Threads : 6 created, 0 imported (0 cached)]
Thread 2 started on node 2
Thread 2 from node 2 finished on node 2: from 0 to 20!
```

Chapter 6

Wizarding PM2: advanced features

$Id: wizarding.tex,v 1.6 2001/04/18 15:13:14 bouge Exp $

We have gathered in this section some advanced material as a reference for the advanced user. Beginners should probably skip this section on a first reading.

6.1 Thread synchronization

PM2 provides a number of POSIX-like synchronization facilities to synchronize the threads within a node and among remote nodes: locks, mutexes, semaphores, conditions, barriers. The facilities which are local to a node are provided by the Marcel thread scheduler. Remote synchronization facilities are provided by PM2.

6.1.1 Locks

The lowest level of synchronization is to temporarily disable the scheduler at the local node. This guarantees that the successive instructions will be issued atomically. This is done using the **`pm2_freeze`**/**`pm2_unfreeze`** routines. These are simply wrappers for two essential Marcel routines respectively called **`lock_task`** and **`unlock_task`**. The **`lock_task`** routine locks a Marcel thread within the scheduler, disabling any external interrupt. No other thread whatsoever can be scheduled until the **`unlock_task`** has been called. These routines may be nested. It is up to the user to guarantee proper nesting. One has to be uttermost careful that PM2/Marcel provides no protection against deadlock. Also, locking a tasked within a node disables any progress of the daemon service threads: no message can be received, etc. It is up to the user to guarantee that no harm may result, for instance from overfilling communication buffers.

Consider Program 6.1. The main thread spawns an even number of threads on the local node. Each even thread increases the shared variable **`counter`** for **`N`** times, and each odd thread decreases it for **`N`** times, too. To emphasize that increasing and decreasing a variable is *not* an atomic operation at the level of machine code, we have written it explicitly

```
#include "pm2.h"

#define N 1000000
#define NB_THREADS 4            // Should be even!

pm2_completion_t c;

volatile int counter = 0;

static void
f (void *arg)
{
  int i;
  int name = (int) arg;
  int forward = ((name % 2) == 0);
  volatile int tmp;
  for (i = 0; i < N; i++)
    {
      pm2_freeze ();

      tmp = counter;
      if (forward)
        tmp++;
      else
        tmp--;
      counter = tmp;

      pm2_unfreeze ();
    }
  pm2_completion_signal (&c);
}
```

Program 6.1: [**`freeze.c`**] Freezing and unfreezing the scheduler (Follows up on page 75).

```
int
pm2_main (int argc, char *argv[])
{
  int i;

  pm2_init (&argc, argv);

  if ((pm2_self ()) == 0)
    {
      pm2_completion_init (&c, NULL, NULL);

      tprintf ("Initial value: %d\n", counter);

      for (i = 0; i < NB_THREADS; i++)
        {
          tprintf ("Creating thread %d\n", i);
          pm2_thread_create (f, (void *) i);
        }

      for (i = 0; i < NB_THREADS; i++)
        pm2_completion_wait (&c);

      tprintf ("Final value: %d\n", counter);

      pm2_halt ();
    }

  pm2_exit ();
  return 0;
}
```

Program 6.1 (Cont'd from page 74): [**`freeze.c`**] Freezing and unfreezing the scheduler.

as a Load/Add/Store series of operations. (Observe that variable **tmp** has be registered as **volatile** to prevent the wise compiler to optimize our trick!). The expected final value for **counter** is 0.

```
ravel% pm2load freeze
Initial value: 0
Creating thread 0
Creating thread 1
Creating thread 2
Creating thread 3
Final value: 0
```

It is interesting to run the program without the calls to **pm2_freeze** and **pm2_unfreeze** routines, and see what happens. At least on my current workstation, this yieds the following result:

```
ravel% pm2load freeze
Initial value: 0
[...]
Final value: 776821
```

It shows that getting interrupted in the middle of the critical section is a relatively frequent event, which should definitely not be overlooked by the PM2 programmer!

6.1.2 Mutexes

Freezing is sufficient for most simple purpose, when a (short!) sequence of actions have to be atomically issued. Yet, this does not handle the case of sequence where the thread may get temporarily suspended, from its own request or not. In this case, more sophisticated facilities have to be used, for instance mutexes (for *mutual exclusion*). Consider Program 6.2, and assume that for some reason, the thread should be suspended right in the middle of the critical section. We simulate this by a call to the **marcel_yield** routine, which yields the control back to the scheduler and inserts back the thread into the ready list. Running the program without any synchronization facility returns a non-null value, even for a few iterations. A mutex has to be first initialized using the **marcel_mutex_init** routine, with a number of attributes. As usual, **NULL** is a sensible default in a first place. A mutex can afterwards be acquired and released, respectively with the **marcel_mutex_lock** and **marcel_mutex_unlock** routines.

6.1.3 Semaphores

Marcel also provides semaphores. A semaphore can be initialized with the **marcel_sem_init** routine with an initial integer value. It can be taken with **marcel_sem_P**, and freed with **marcel_sem_V**, as (probably?) expected.

```
#include "pm2.h"

#define N 100
#define NB_THREADS 4            // Should be even!

pm2_completion_t c;

volatile int counter = 0;

marcel_mutex_t mutex;

static void
f (void *arg)
{
  int i;
  int name = (int) arg;
  int forward = ((name % 2) == 0);
  volatile int tmp;

  for (i = 0; i < N; i++)
    {
      marcel_mutex_lock (&mutex);

      tmp = counter;
      marcel_yield ();
      if (forward)
        tmp++;
      else
        tmp--;
      counter = tmp;

      marcel_mutex_unlock (&mutex);
    }
  pm2_completion_signal (&c);
}
```

Program 6.2: [**mutex.c**] Using mutexes in PM2 (Follows up on page 78).

```c
int
pm2_main (int argc, char *argv[])
{
  int i;
  pm2_init (&argc, argv);
  if ((pm2_self ()) == 0)
    {
      pm2_completion_init (&c, NULL, NULL);
      tprintf ("Initial value: %d\n", counter);

      marcel_mutex_init (&mutex, NULL);

      for (i = 0; i < NB_THREADS; i++)
        {
          tprintf ("Creating thread %d\n", i);
          pm2_thread_create (f, (void *) i);
        }

      for (i = 0; i < NB_THREADS; i++)
        pm2_completion_wait (&c);

      tprintf ("Final value: %d\n", counter);
      pm2_halt ();
    }
  pm2_exit ();
  return 0;
}
```

Program 6.2 (Cont'd from page 77): [**`mutex.c`**] Using mutexes in PM2.

```
#include "pm2.h"

#define N 100
#define NB_THREADS 4             // Should be even!

pm2_completion_t c;
volatile int counter = 0;
marcel_mutex_t mutex;
marcel_sem_t sem;

static void
f (void *arg)
{
  int i;
  int name = (int) arg;
  int forward = ((name % 2) == 0);
  volatile int tmp;

  for (i = 0; i < N; i++)
    {
      marcel_mutex_lock (&mutex);

      tmp = counter;
      marcel_yield ();
      if (forward)
        tmp++;
      else
        tmp--;
      counter = tmp;

      marcel_mutex_unlock (&mutex);
    }

  marcel_sem_V (&sem);
}
```

Program 6.3: [**`semaphore.c`**] Using semaphores in PM2 (Follows up on page 80).

```
int
pm2_main (int argc, char *argv[])
{
  int i;
  pm2_init (&argc, argv);
  if ((pm2_self ()) == 0)
    {
      marcel_sem_init (&sem, 0);

      tprintf ("Initial value: %d\n", counter);
      marcel_mutex_init (&mutex, NULL);

      for (i = 0; i < NB_THREADS; i++)
        {
          tprintf ("Creating thread %d\n", i);
          pm2_thread_create (f, (void *) i);
        }

      for (i = 0; i < NB_THREADS; i++)
       marcel_sem_P (&sem);

      tprintf ("Final value: %d\n", counter);
      pm2_halt ();
    }

  pm2_exit ();
  return 0;
}
```

Program 6.3 (Cont'd from page 79): [**`semaphore.c`**] Using semaphores in PM2.

In fact, completion are just a simple wrapping for semaphores. On initializing a completion, a semaphore is initialized, and its address and native node are stored into a structure. Waiting for a completion is just issuing a **marcel_sem_P** on the semaphore. Signaling completion includes the following action: if the local node is the same as the native node, then a call to **marcel_sem_V** is issued for the semaphore; otherwise, a RPC request is issued to the native node, with the the signaling routine as its service and the completion as its argument. Program 6.3 is the same as Program 6.2, but completions have been replaced by explicit semaphores.

6.1.4 Conditions

Marcel finally provides conditions variables associated to mutexes, with a Mesa-style semantics. A thread which signals a condition keeps on running. One of the threads blocked waiting for this condition is selected and appended to the queue of the threads ready to run. As a consequence, *user cannot be guaranteed that a thread which exits a wait is immediately executed*. Actually, no hypothesis may be made about the state of the thread at the time it is eventually scheduled. This is the reason why it is necessary to encapsulate waiting commands within **while** loops: the first action of a thread which has been signaled is then to retest the condition which commanded its waiting; if it does not hold, then it goes back to sleep, waiting again to be signaled.

In Program 6.4, we have further constrained the shared counter to take values only between **MIN_COUNTER** and **MAX_COUNTER**. This is done using two condition variables to control increment and decrement, similarly to a classical producer-consumer scheme. (We have reverted to direct operations on the **counter** variable to enhance readability.)

A condition is declared using the **marcel_cond_init** routine. It takes as arguments the condition and an attribute. As usual, **NULL** is a sensible default for the attribute. Waiting on a condition makes sense *only* within a critical section controlled by some mutex. The mutex is passed to the routine **marcel_cond_wait** as an additional parameter. The very first job of this routine is to acquire the mutex. In the regular use of condition variables, the waiting routine is called within the critical section controlled by this very mutex, so that is it necessarily free. But it is not mandatory to do so, though the writer cannot see any sensible example of such a situation at this time... The condition is signaled using the **marcel_cond_signal** routine. Observe this does not require any reference to the controlling mutex. This routine wakes up exactly one waiting thread, if any exists. Alternatively, the **marcel_cond_broadcast** routine wakes up *all* threads waiting on the given condition. It is also possible to specify a limit on the time a thread should wait for a condition, using the **marcel_cond_timedwait** routine. In this case, the routine returns a **ETIMEDOUT** value if the thread has not been signaled within the specify delay.

NB *LB to RN: Please check carefully!*

```c
#include "pm2.h"

#define N 100
#define NB_THREADS 4            // Should be even!
#define MAX_COUNTER 2
#define MIN_COUNTER -2

pm2_completion_t c;
volatile int counter = 0;
marcel_mutex_t mutex;
marcel_sem_t sem;
marcel_cond_t forward_enabled, backward_enabled;

static void
f (void *arg)
{
  int i;
  int name = (int) arg;
  int forward = ((name % 2) == 0);

  for (i = 0; i < N; i++)
    {
      marcel_mutex_lock (&mutex);

      if (forward)
        {
          while (counter >= MAX_COUNTER)
           marcel_cond_wait (&forward_enabled, &mutex);
          counter++;
          tprintf ("%d\t%d\n", counter, name);
          marcel_cond_signal (&backward_enabled);
        }
      else
        {
          while (counter <= MIN_COUNTER)
           marcel_cond_wait (&backward_enabled, &mutex);
          counter--;
          tprintf ("%d\t%d\n", counter, name);
          marcel_cond_signal (&forward_enabled);
        }

      marcel_mutex_unlock (&mutex);
    }
  marcel_sem_V (&sem);
}
```

Program 6.4: [**condition.c**] Using condition variables in PM2 (Follows up on page 83).

```
int
pm2_main (int argc, char *argv[])
{
  int i;
  pm2_init (&argc, argv);
  if ((pm2_self ()) == 0)
    {
      marcel_sem_init (&sem, 0);

      tprintf ("Initial value: %d\n", counter);
      marcel_mutex_init (&mutex, NULL);
      marcel_cond_init (&forward_enabled, NULL);
      marcel_cond_init (&backward_enabled, NULL);

      for (i = 0; i < NB_THREADS; i++)
        {
          tprintf ("Creating thread %d\n", i);
          pm2_thread_create (f, (void *) i);
        }

      for (i = 0; i < NB_THREADS; i++)
        marcel_sem_P (&sem);

      tprintf ("Final value: %d\n", counter);
      pm2_halt ();
    }

  pm2_exit ();
  return 0;
}
```

Program 6.4 (Cont'd from page 82): [**condition.c**] Using condition variables in PM2.

6.2 Profiling PM2 programs

NB *LB: Yet to do.*

6.3 Miscellaneous

6.3.1 Application initialization with PM2

NB *LB: Yet to do. Describe* `pm2_set_startup_func`*?*

Chapter 7

Additional PM2 material

$Id: appendicing.tex,v 1.4 2001/04/18 15:13:12 bouge Exp $

7.1 Using a standard main

Some application may require to use the traditional **main** function name instead of the regular **pm2_main**. This can be enabled through the **standard_main** options of the Marcel flavors. This option defines the **STANDARD_MAIN** compilation flag, tobe used for each of the PM2 programs. Note that this may affect the performance of several thread management functions. The user is therefore discouraged to do this unless explicitly required.

7.2 Synopsys of PM2 scripts

7.2.1 pm2conf

```
ravel% pm2conf -h
Usage: pm2conf { 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% pm2conf -s tcp cluster0 -s tcp -e cluster 10-12
0 : cluster0tcp
1 : cluster10tcp
2 : cluster11tcp
3 : cluster12tcp
```

7.2.2 `pm2load`

```
ravel% pm2load -h
Usage: pm2load { 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)
-c name: Use console named "name"
-l: Also generate a log file on first node
-L: Only generate a log file on first node
-h: Display this help message
```

NB *LB to RN: The -L option does not seem to work properly*

7.3 Common Pitfalls

Table 7.1 sums up the symptoms together with possible solutions.

7.4 Frequently asked questions

Why do I get the message ”Permission denied” when running pm2load?

Why does my program crash the entire cluster?

Table 7.1: Common Pitfalls

Symptoms	Solutions
`Permission denied` error at load time	Each configuration node must be made accessible by **`rsh`** from the local host: update your **`.rhosts`** file

Chapter 8

Reading on PM2

$Id: reading.tex,v 1.3 2001/04/18 15:13:14 bouge Exp $