Simulation of the communication libraries of the CM-5 on UNIX workstations

Nicolas Guérin and Jean-Luc Gaudiot

CENG Technical Report 95-19

Department of Electrical Engineering - Systems
University of Southern
Los Angeles, California 90089-2562
(213)740-4484

&

École Nationale Supérieure de l'Aéronautique et de l'Espace
31055 Toulouse - FRANCE

August 25, 1995

Abstract

The purpose of the project described here was to build a function library on UNIX workstations, that would emulate the communication libraries of the CM-5: CMMD and CMAML. This simulation should enable the user to compile, run and debug CM-5 programs on a network of UNIX workstations.

The first part of this report is a general presentation of the CM-5 and its communication libraries: architecture, execution model, active messages etc.

Then, you will find in the second part the details of the implementation: how the simulation is working, how UNIX communicates with UDP/IP, why an acknowledgment scheme was necessary, how it is implemented, and the limitations of the simulation.

Finally you will be given the performances of the simulation as compared to a CM-5, the influence of the number of nodes and the interest of the long active message scheme.

The Appendix contains the fully commented listing of the whole program.

ACKNOWLEDGMENT

I would like to express my gratitude to Dr. Jean-Luc Gaudiot, Associate Professor at USC, and to Steve Jenks, Ph.D. student and advisor of my research work, for their guidance and support throughout the five months of my training session at USC.

I am also very grateful to Dr. Bernard Lécussan, Chairman of the ENSAE Computer Science Department, who made all this possible.

This research was conducted as a requirement from Sup'Aéro for obtaining the ENSAE Engineering Diploma.

TABLE OF CONTENTS

ABSTRACT 2

ACKNOWLEDGMENT 3

TABLE OF CONTENTS 4

INTRODUCTION 7

A - BACKGROUND 8

I - The CM-5 .... **8**
1 - CM-5 Architecture 8
2 - The CM-5 execution model 9
3 - Languages 10

II - The communication libraries of the CM-5 .... **11**
1 - CMMD 11
a- General presentation 11
b- CMMD timers 11
2 - CMAML: CM Active Message Layer 12
3 - Handler functions 13
4 - Active messages 13
5 - Requests, Replies and RPCs 14

III - Functions to implement **15**
1 - CMAML functions 15
a- Functions for sending active messages: 15
b- Network polling functions 15
2 - CMMD functions 16
a- CMMD timers 16
b- Other functions 16
c- For compatibility only 16

B - IMPLEMENTATION 17

I - General presentation of the program **17**
1 - The package 17
2 - Language 18
3 - Installation - Compilation of the library 18
4 - The "HOSTS" file 18
5 - Launching sequence 19
6 - Modifications in the existing program 20
7 - Example 20

II - Communications **21**
1 - sockets 21
2 - Protocols 21
3 - Non-blocking mode 22

III - The acknowledgment scheme **23**
1 - Why ? 23
2 - The ACK scheme 23
a - First implementation 23
b - The "acknowledgment size" 24
c - Current implementation 25
d - Limitation 25
3 - How to choose the "acknowledgment size" 26
4 - Implementation of an ACK message 27

IV - Active messages **27**
1 - Normal active message 27
a - Composition 27
b - Implementation 28
2 - Long active message 28
a - Presentation 28
b - Implementation 29
c - Possible improvement 29

V - The launcher **30**
1 - Presentation 30
2 - Launching processes 30
a - Command-line arguments 30
b - Reading the data from file "HOSTS" 30
c - Launching the processes 31
3 - Synchronization 31
4 - Detection of the end of the program 32
5 - Options 33

VI - Initialization of the node **34**
1 - Modification of the CMMD program 34
2 - CMMD_init_simulation() 35

VI - Functions implementation **36**
1 - C++ implementation: the "node" class 36
2 - Timers 38
3 - CMMD_sync_with_nodes() 38
4 - CMAML_poll() 38

5 - CMAML_request(), CMAML_reply(), CMAML_rpc() 39
6 - Others 40
a - CMMD_self_address() 40
b - CMMD_partition_size() 40
c - CMMD_disable_interrupt(), CMAML_disable_interrupt(), CMMD_fset_io_mode()40
VII - Limitations **40**
1 - Homogeneous network 40
2 - Number of nodes 41

C - PERFORMANCES 42

I - Benchmarks **42**
1 - The "nt-tak" program 42
a - Presentation 42
b - Implementation 42
c - features 42
d - Results 43
2 - The "nt-pmm" program 43
a - Presentation 43
b - Implementation 43
c - features 43
II - The platform issue **44**
III - Influence of the number of nodes **45**
1 - Time to complete 45
2 - Overall efficiency 46
IV - Using long active messages **47**
V - Comparison with the CM-5 .... **48**

CONCLUSION 49

BIBLIOGRAPHY 50

APPENDIX 51

Listing of the simulation **52**
Listing of the example **75**

INTRODUCTION

The purpose of this project was to build a function library on UNIX workstations, that would emulate the communication libraries of the CM-5. This should enable the user to compile and run CM-5 programs on a network of UNIX workstations.

This project has been conducted for several reasons.

The main one is that having a CM-5 simulator may be useful for debugging programs when the real machine is unavailable or when running the program on the real hardware causes deadlocks. It should be easier to debug programs on a workstation acting like a CM-5 than on the actual machine.

Besides, this is my end of study project, and thus part of my engineering studies; building a simulator of the CM-5 seemed to be an excellent way of learning the message passing and execution models of distributed memory parallel computers. In addition, it gave me a good idea of how programs are structured for such machines.

For those reasons, I found this project very challenging and interesting, and this report presents the results of my work.

A - BACKGROUND

I - The CM-5

The CM-5 system from Connection Machine is a multiprocessor system designed to achieve high performance on large and complex problems. This section will briefly describe the architecture of the CM-5 system, and its communication libraries. Most of this information comes from [1].

1 - CM-5 Architecture

A CM-5 system consists of a large number of processing nodes, a small number of partition managers (PMs) and some number of I/O control processors (IOCPs) and I/O devices. These components are all linked together by two internal communications networks, the Data Network and the Control Network.

The Data Network is used for fast, high-bandwidth communication of data between the processing nodes, PMs, and I/O devices. The Control Network provides a number of global control operations, such as broadcast, scan/reduction operations, and node synchronization.

In the CM-5 system, each processing node contains a RISC microprocessor and a Network Interface (NI) chip that connects it to the networks. Each processing node may also contain four vector processor units (VUs), which are located between the RISC microprocessor and the node's memory. The VUs, when they are present, provide highly efficient memory-based arithmetic calculations.

The following figure shows the components of a CM-5 processing node:

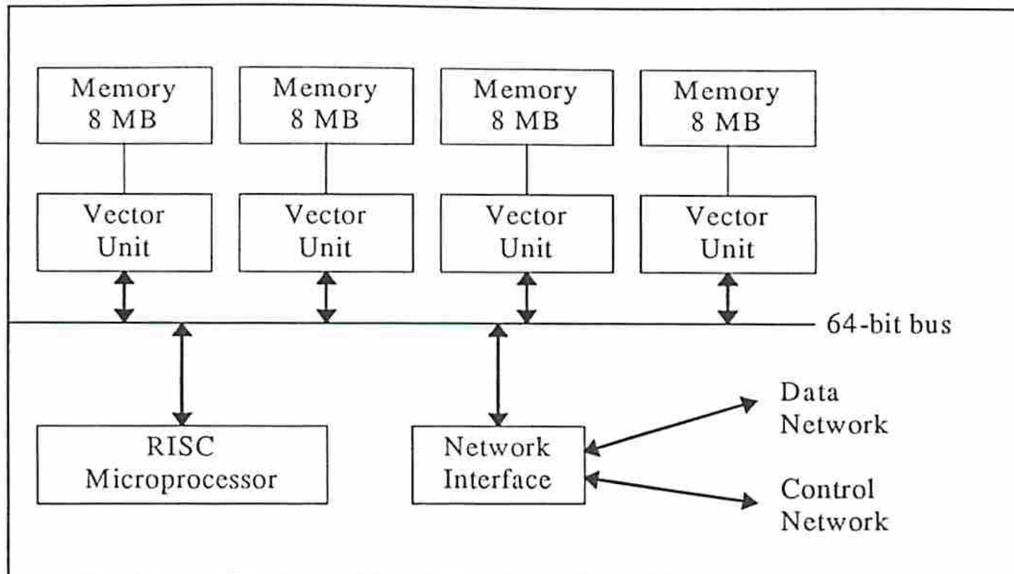

Figure 1. Components of a CM-5 processing node

The CM-5 processing nodes are grouped into one or more partitions, each controlled by a partition manager. Program execution always begins and ends on the partition manager.

The CM-5 system uses a parallel timesharing operating system known as CMost, which allows each partition of nodes to execute as a separate parallel timesharing system. Each partition has its own set of processes to execute, and can be considered a separate parallel processing system.

Each node within a partition has a logical address relative to that partition, from 0 to (partition-size -1). The nodes (and the host) use these logical addresses when sending messages. They use specific functions to gain the requisite information, among which CMMD_self_address() and CMMD_partition_size(), that will have to be implemented in the simulation.

2 - The CM-5 execution model

An application program executing on the CM-5 is actually two separate programs. One program runs on the partition manager (the "host" in CMMD terminology); the other runs on all nodes. Frequently, this duality is transparent to programmers: a programmer simply writes a global application, or a "hostless" message-passing program, and the CM software does the work required to create the appropriate host and node executables. The simulation, as it will be described later, is working exactly the same way, using the same duality between the "host" program and the node program.

In a CM-5 message-passing program:

- The program executes in multiple copies, one on each processing node. Although all nodes execute the same program, each node operate independently of the others.

- Communication between nodes is in the form of “messages” sent from one node to another, or from one node to many nodes. It is with theses messaging features that the CMMD library, and thus the simulation, is mainly concerned.

- The partition manager typically does nothing more than start the program running on each of the nodes, and thereafter acts as an I/O server for the nodes. This is the standard, “hostless” form of message passing.

- Alternatively, the partition manager may execute a separate “host” program that directs the actions of the nodes, distributes data to them, and collects results. This is the “host/node” form of message passing. This form is not implemented in the simulation, and therefore won’t be described much more. For more information on the host/node programming, see [1].

- I/O may be handled by each node independently, with each node opening and closing its own files.

3 - Languages

CMMD programs can be written in standard Fortran 77, C, or C++. Programs written in these languages execute exclusively on the microprocessor in each node, ignoring the vector units. The simulation, however, only supports C++, and thus C. Nevertheless, it should be possible to call the simulation functions from a Fortran program, but it would imply some work on the Fortran code.

CMMD programs can also be written in the CM’s data parallel programming languages, CM Fortran and C*, allowing them to access the vector units. The simulation does not support those languages.

II - The communication libraries of the CM-5

1 - CMMD

a- General presentation

CMMD is a library of message-passing routines for the Connection Machine CM-5 system. Programs that use CMMD typically operate in a message-passing style:

- Each processing node of the CM-5 runs an independent copy of a single program, and manages its own computations and data layout.
- Communication between nodes is handled by calls to CMMD message-passing functions

CMMD allows you to send messages from one processing node to another in a number of different ways, depending on the needs of your application. Therefore, the CMMD library provides a wide range of communication functions, and only a few of these functions are actually implemented in the simulation. What is, and what is not implemented will be fully described later in the report.

b- CMMD timers

The CMMD library provides a set of timing routines for use by the nodes in message-passing programs. This routines are implemented in the simulation, and provide a good way to evaluate the performances of the simulation and compare them to the CM-5.

Each node manages its own timers. The timers are called by a single node and record the time spent by that node. They have no connection with timers used on any other node.

These timers measure three values:

- Busy time is the time during which the user program is executing user code.
- Idle time is the time during which the user program is looping in the operating system's dispatch loop.
- Elapsed time is the sum of busy time and idle time.

Each node may have up to 64 timers running simultaneously, and timers can be nested.

All timers share the same pattern of use:

- First, you must call CMMD_node_timer_clear() with an integer timer-ID. This creates the timer and initializes it to zero.
- Then you call CMMD_node_timer_start(), to start the timer going.
- The next call should be CMMD_node_timer_stop(). This records the current value within the timer.
- You may call CMMD_node_timer_start() again, timing will be cumulative until you call CMMD_node_timer_clear() again.

2 - CMAML: CM Active Message Layer

CMAML is the protocol-less transport layer upon which the higher-level CMMD functions are built. You can use these functions to define custom network protocols and to perform specialized low-level communication.

CMAML is an internal layer of the CMMD software. It is made available for advanced programmers and library developers who may require functionality or performance not otherwise provided by the higher level CMMD functions.

Actually, the purpose of the simulation was to provide CMAML functions, more than CMMD functions. This is the reason why it is important to describe this library further.

CMAML provides transport functions for two types of message transfer:

- Active messages are one packet long (five 32 bits words). Their purpose is to invoke a function upon the receiving node and to supply arguments to be passed to the function upon invocation.
- Block data transfers, transfer a single block of data from one or more source nodes to a single receiving node. When the transportation operation completes, the source and the destination nodes may invoke an attached handler function. This type of transfer was not needed and therefore is not implemented in the simulation. It is however possible to implement it on the existing simulation source code. Block Data Transfers will not be described here. For more information, see [1].

In support of these transport functions, CMAML also provides functions that allow polling of the networks, manipulating interrupt states, and preserving register contents.

As with most transport layers, CMAML functions assume that some higher layer of software is providing any protocol needed, and that (for example) receiving nodes know

what to do with any data sent to them. Users must ensure that their applications provide such protocol.

3 - Handler functions

A handler is simply a user-defined function that can be invoked in any one of a number of situations:

- upon receipt of an Active Message
- upon receipt of all expected data on a receive port
- upon completion of CMAML_scopy() or CMAML_pcopy() operations
- upon completion of CMMD_send_async() or CMMD_receive_async() operations

Only the first situation is used in the simulation, since the functions used in the other situations are not implemented.

Note that handlers are, by definition, executed upon completion of some event, not upon attaining a given line of code in a program. They thus execute asynchronously to the application's main thread of control. In fact, they may cause intermissions in the background computation whenever the application allows it. Handlers, therefore, must be written so as not to interfere with the state of the background computation. The user is responsible for the correct interaction between the background computation, communication, and handlers.

4 - Active messages

An active message is a single network packet comprising a series of words. The first word (word 0) is the address of a handler function on the destination node. The remaining words are arguments to be passed to the function.

The current CMAML implementation uses a five word format:

Function Address	arg1	arg2	arg3	arg4
word0	word1	word2	word3	word4

Figure 2. Active message format

Receipt of an active message automatically invokes the handler function with the specified arguments. The interpretation of the arguments is strictly under the control of the handler function. When the handler function returns, the node either resumes its polling of messages or, if it was interrupted to receive the message, continues the execution of its interrupted computation.

Active messages are never buffered on arrival. When an active message is received (whether by polling or by interrupt), its handler is invoked immediately.

5 - Requests, Replies and RPCs

There are three types of active messages: requests, replies and RPCs. They differ in their use of the data network, and in the context in which they can be used.

The Data Network has two independent interfaces. They are sometimes called the "request" and "reply" interfaces.

Here are a few differences between the three types of active messages.

A request message

- Can only be sent from the main thread of control, never from a handler.
- Uses the request interface to the Data Network to transmit data.
- May send only Reply messages from within its handler.
- Polls both interfaces for incoming messages.

A Reply message:

- Can only be sent from handler functions, never from the main thread of control.
- Uses the reply interface to transmit data
- Cannot call any communication functions from within its handler.
- Polls only the reply interface for incoming messages.

A Remote Procedure Call (RPC) message:

- Can be sent either from a node's main thread of control or from a handler function.

- Uses the request or reply interface to the Data Network to transmit data.
- Can call other RPCs, Replies, or any other non-blocking function.
- Polls both interfaces for incoming messages.

Request and Reply messages are often used for purposes where one node sends a Request for data, and the other returns the data as a Reply.

In spite of their differences, Request and RPC messages are the same for the simulation. Since there is no way to control if the message has been sent from a node's main thread of control or from a handler function, and since RPC messages may use either interfaces to transmit data, there is no problem in implementing RPC messages as Reply messages.

III - Functions to implement

Here is the list of the functions that had to be implemented. A better description of these functions will be given later in paragraph B.

1 - CMAML functions

a- Functions for sending active messages:

```
void CMAML_request
void CMAML_reply
void CMAML_rpc
            (int dest_node, void (*handler)(), int data1, int data2, int data3, int data4)
```

dest_node	Integer specifying the destination node for the message.
handler	Address of a handler function on the destination node.
data1..4	Handler function arguments.

b- Network polling functions

```
void CMAML_poll()
void CMAML_request_poll()
```

```
void CMAML_reply_poll()
```

CMAML_poll() is the generic CMAML polling function, and checks both interfaces for messages.

2 - CMMD functions

a- CMMD timers

```
int CMMD_node_timer_clear(int timer)
int CMMD_node_timer_start(int timer)
int CMMD_node_timer_stop(int timer)
double CMMD_node_timer_elapsed(int timer)
double CMMD_node_timer_busy(int timer)
double CMMD_node_timer_idle(int timer)
```

timer Integer from 0 to 63, identifying the timer.

b- Other functions

```
int CMMD_self_address()
```

This function give the logical address of the node in its partition.

```
int CMMD_partition_size()
```

Returns the number of nodes in the partition.

```
void CMMD_sync_with_nodes()
```

This function allows a synchronization between the nodes. Once a program on a specified node calls this function, it waits until all the other nodes also call this function.

c- For compatibility only

```
void CMMD_disable_interrupts()
int CMMD_fset_io_mode(File *,int)
```

Those functions have no effect on the simulation, and are provided only to avoid compatibility problems when compiling a CM-5 program for the simulation.

B - IMPLEMENTATION

I - General presentation of the program

1 - The package

While developing this project, it was important to keep in mind that the potential user of the simulation would probably be familiar with the CM-5 interface, and thus it seemed important to keep a very simple interface. A program running on the CM-5 should be able to run on the simulation with only minor modifications.

And in fact, except for an indispensable initialization, the CM-5 programs do not need any modification. Once the compilation options have been modified to provide a link edition with the library, the program may compile as if it were on a CM-5. The only difference is that, when the program calls a CMAML function, the functions from the library will be used instead of the genuine CM-5 functions.

Once the simulation has been compiled, only three files are useful:

- cmmd.h : This file replaces the original cmmd.h of the CM-5. It contains the definition of the functions available in the library. You can also take a look at this file to know the exact syntax and types of the functions.

- libcmmd.a : This is the library itself. You have to specify a path to this library when compiling and binding a CMMD program.

- go : As its name suggests, this program is the launcher of the simulation. It runs the program on the workstations specified, and also acts like the partition manager of the CM-5: it is for example used for synchronization.

When running your CM-5 program, the simulation also needs another important file:

- HOSTS : This file contains major information for the simulation: the hosts for each node, the login name on each host, the ports to use, the path and the name of the executable.

The package also contains an example of CMMD program. See 7- for more information about it.

2 - Language

The simulation has been programmed using the C++ programming language. For this reason, it will handle perfectly programs written in C++ for the CM-5. This language has been chosen because it provides a good flexibility, and can still be very efficient thanks to the C functions it is able to call.

Programs written in C can be easily adapted, since C is an internal layer of C++. Such programs only need to be compiled with a C++ compiler, and linked to the library. In the same way, programs written in Fortran 77 should be able to use the library, provided that the compiler allows calls to C++ functions.

3 - Installation - Compilation of the library

The first thing you need to do to install the library is to "un-tar" the file "CMMDsim-1.0.tar". This will create the directory "CMMDsim-1.0" and five sub-directories: "src", which contains the source code of the library, "example", which contains an example of CMMD program, "bin", "lib" and "include".

Compiling the library should be really easy. It has been programmed on hp and SUN workstations, but there is no reason why it would not run on other UNIX platforms. It is possible to use the "g++" compiler from the GNU C compiler (gcc), but other compilers are also fine (for example hp's CC compiler).

Just type "make all" in the "src" directory: every file will be compiled, including the **go** program which is the only executable, and the library "libcmmd.a" is created. The simulation is then ready.

4 - The "HOSTS" file

As it has been already said before, this file contains major information for the simulation: the hosts for each node, the login name on each host, the ports to use, the path and the name of the executable.

Here is an example of HOSTS file:

```
# Host      port    Command   Login    Name and  Location of binary file
hyde        12365    remsh    guerin   main      /users/guerin/project/example
chryse      10248    remsh    guerin   main      /auto/users/guerin/project/example
```

When you want to run the simulation, run the **go** program; it will first read the HOSTS file, then launch the node processes on each specified host.

In the example given, **go** will launch two processes, one on the host "hyde", the other on "chryse". The program launched is called "main". It will use the ports 12365 to 12370 on hyde, and the ports 10248 to 10253 on chryse. Each process uses indeed five ports. This specification of the port is useful when one of the ports you intended to use is already occupied. You just have to modify the number in the HOSTS file, and try again.

The executable is not located in the same directory on each host, and this is why you have to specify it for each node.

You can also use a different login for each host. This is useful when you have several logins, or when you want to use a machine with the login of a friend. Nevertheless, you must be allowed to do so! (e.g. your login must be in the ".rhosts" file). For more information, read the "rsh" or "remsh" man pages. Indeed, the name of the "remote shell" UNIX program is not the same on every platform: "rsh" on SUN, "remsh" on hp; that is why you have to specify it in the HOSTS file.

You can also launch several processes on a single machine. It is not recommended though: socket communications among a single machine is not very reliable. And furthermore, this will slow down the simulation.

There is no "partition" when you use the simulation. The partition size is just the number of entries in the HOSTS file. There is no explicit limitation in the number of nodes you might use. Nevertheless, you should read VII- before trying to run the simulation on ten thousand hosts.

5 - Launching sequence

This is the sequence of actions that happens when the user types **go**:

- The program reads the command-line arguments, in order to feed them back to each node
- **go** reads the "HOSTS" file
- for each entry in this file, a process is launched with its number in the command line: There is no other way for a process to know its logical address (CMMD_self_address()). Hence, we have to modify the existing program (see 6- for more details).
- When all processes are launched, **go** waits for an acknowledgment from all of them. This allows the simulation to synchronize. It is done by a call to the CMMD_sync_with_nodes() function.

For more technical details, see V-.

6 - Modifications in the existing program

Several parameters have to be given to each process, among which the indispensable logical address. Those parameters are given among the command-line arguments of the process. Therefore, the existing program has to be slightly modified.

A special function was created to handle the command-line arguments: CMMD_init_simulation(). In spite of its name, it is not a real CMMD function. It has been created for the simulation only. Not only does it handle the command-line arguments, but it also initiates the communications.

The only modification that you need to do in the existing program is to add these few lines just after the declaration of the main() function:

```
// to add if you are using the simulation; needs stdlib.h (atoi())
// begin
{
  // test number of arguments
  if (argc<6) {
    exit(2);
    }

  // Init with the 5 last arguments
  CMMD_init_simulation(atoi(*(argv+argc-5)),*(argv+argc-4),atoi(*(argv+argc-3)),
      atoi(*(argv+argc-2)),atoi(*(argv+argc-1)));

  // Ignore the 5 last arguments
  argc-=5;
}
// end
```

The "argc" and "argv" variables must be declared in the definition of main().

7 - Example

An example of CMMD program is included with the package (directory "example"). It is very simple and it is probably not necessary to explain what it does. It has been provided to familiarize the user with the simulation, before he starts to run much more complicated programs.

First, you must compile it (see the Makefile), and then you can try to run it. Don't forget to modify the "HOSTS" file for your system. If you have any problem running it, the answer is probably in this report.

II - Communications

This project requests the use of the communication tools of UNIX. To implement the communication functions, it is necessary to use low-level UNIX functions, and to use the sockets. Here is a brief and simple description of UNIX communication tools. For more information on UNIX communications, see [2].

1 - sockets

A socket is a very powerful UNIX tool that allows you to exchange messages with another host. A socket is working like a letterbox: it has an address and a buffer. A socket address is defined by its host and a port number. If you send a message through a socket, you only have to specify the address of the destination socket. When a message is received, you can also know the address of the sender. There is a buffer where messages are stocked, waiting for the receiver to read them.

2 - Protocols

You can use two different protocols when using UNIX sockets: UDP/IP and TCP/IP, which is the best known since it is used on the Internet.

TCP/IP has a big advantage: it is safe; a message can not be lost, which means that each message sent is indeed received. In fact, a TCP/IP communication, or *stream* communication, needs the sockets to be in a *connected* mode. Each socket is linked to exactly one other, and can send or receive messages only with this socket.

UDP/IP is working more like a mailbox: when a message is sent, it is put in a FIFO queue, and the receiver may or may not read the message. This is known as the *datagram* mode. A socket can send a message to any other socket, provided that it knows its address. The sockets do not need to be *connected*. Furthermore, a message can be lost: there is no way to be sure that a message has been received.

Even if TCP/IP offers reliability, UDP/IP has been chosen for this project. These are the two main reasons:

- UDP/IP is faster than TCP/IP.
- You don't want to saturate the ports of the machine. The connected mode needs a lot of sockets, since a socket can only communicate with one other. If N is the number of nodes you want to use, you will need at least 2N sockets on each host (because there are two interfaces). Thus, if you use 32 nodes, you will need 64 sockets, whereas you need only 5 sockets for the datagram mode.

The fact that you can lose messages with UDP/IP is a real problem, and you will see in III- how the simulation copes with it.

3 - Non-blocking mode

Many of the CM-5 communication functions must not block. For example, a call to CMAML_poll() should return quickly if there is no message pending. This means that the simulation has to use communication functions in non-blocking mode, which is not the default mode.

Furthermore, the simulation sometimes needs to be in blocking mode. For example when sending a message, you do not want it to be truncated, and you want to send the whole message. For this, you need to be in blocking mode.

The simplest way to do this is to use the **ioctl** library, which provides tools to handle this problem. When applying the ioctl function on the socket file descriptor with the FIONBIO option and the proper flag, you can switch between blocking and non-blocking mode.

Here are the functions used in the simulation (file **node.cc**):

```
// ---------------------------------------------------------------------
// - setsocknoblock() - Set socket in non-blocking mode -
// ---------------------------------------------------------------------

static inline int setsocknoblock(int fd) { //fd = file descriptor
  int flag = 1;
  return(ioctl(fd,FIONBIO,&flag));
}


// ------------------------------------------------------------
// - setsockblock() - Set socket in blocking mode -
// ------------------------------------------------------------

static inline int setsockblock(int fd) { //fd = file descriptor
  int flag = 0;
  return(ioctl(fd,FIONBIO,&flag));
}
```

III - The acknowledgment scheme

1 - Why ?

The UNIX protocol used in this simulation is UDP/IP. With this protocol, there is no guaranty that the message is indeed delivered. Are messages lost often ? sometimes ? hardly ever ? never ?

A benchmark program was necessary to answer these important questions. This program allows two hosts to exchange messages: one host sends n messages, and the other receives m. Is m equal to n ? Here are the results.

When 1000 messages are sent, 1000 messages are received. Great ! no loss !

But when sending 100,000 messages, between 2% and 6% are lost, i.e. between 2000 and 6000! This is very embarrassing... And it proves that an active message can be lost quite often in the simulation.

I ran the benchmark with the two nodes being on the same host. The percentage of messages lost jumped to 12%. This is one of the reasons why it is not a good idea to run several nodes on a single host...

At first, I ignored this situation. But when I ran a CMMD benchmark program that exchanged many active messages between nodes, I obtained very strange behaviors because messages were lost. It thus appeared to be a major problem, and the only solution was to implement an acknowledgment scheme.

2 - The ACK scheme

a - First implementation

The first implementation was simple:

- Every message has a number.
- Each time a node receives a message, it sends an acknowledgment.
- Each time a node wants to send a message, it waits until the previous message sent to the same node is acknowledged. If it is not after some time, the node sends again the previous message (which as not been acknowledged) and returns in the loop to receive an acknowledgment.
- If a node receives an acknowledgment that was already received previously, then it just ignores it.

- If a node receives a message for the second time, it must send an acknowledgment (the previous was probably lost), but it must not execute the handler for a second time.

This scheme worked perfectly with the benchmark that used to fail. The problem was resolved... but the new simulation was twice as slow ! Indeed, the number of messages exchanged between nodes is doubled, and the nodes have to wait for an acknowledgment when they need to send a message.

b - The “acknowledgment size”

A simple way to control the time wasted by a node waiting for an acknowledgment is to reduce the frequency of those verifications. For example, a node could send an acknowledgment every twenty messages instead of every message. This would reduce the time spent with acknowledgment by twenty since the nodes verify and wait for an acknowledgment only once every twenty messages.

The new specifications are as follow:

Let “N” be the “acknowledgment size”

- Each time a node receives a message, it increments a counter (specific to the sender).
- When a counter reaches N:
  - It is reset.
  - An acknowledgment is sent to the sender.
- Each time a node sends a message, it first increments another counter (also specific to the receiver).
- If the counter is less than N, the message is sent.
- if it is equal to N:
  - It is reset.
  - The node waits for an acknowledgment. If it is received, the message is sent.
  - Otherwise, the node sends all N previous messages again, and waits for an acknowledgment. This is a loop; the node does it again until it receives the acknowledgment. Then the message is sent.

- Else, the message is sent straight away.
- Finally, the message is stored (in case it has to be sent again).

This scheme is a bit more elaborate, and not only does it work perfectly, but the time lost is almost negligible, compared to the version without acknowledgment.

c - Current implementation

I thought that this scheme was sufficient to cope with every message lost. But then, I tried another CMMD benchmark, and it did not work well. Here is why:

I worked with two nodes. The main characteristic of this benchmark was that when a node sends a message, it waits immediately for the answer, in a blocking loop. If this message was lost, there was no way to exit the loop. And since the simulation is supposed to work without modifying the original program, the acknowledgment scheme had to be modified.

To solve this problem, it is necessary to add a new scheme to the previous one:

- Each time a node *polls* for messages, a counter is incremented.
- Each time the node sends a message, the counter is reset.
- When the counter reaches M (number defined by the user, about 1000 or more), it means that the node has been polling for M times in a row, without sending any message. Then
    - The counter is reset.
    - The node sends again <u>all</u> the messages that have not been acknowledged: reply or request messages, to every node. The message that had been lost is thus sent again, and the program may continue.

This is probably slowing down the simulation, since there might be many messages to send. Nevertheless, it is hardly noticeable, and thanks to this scheme, the simulation is now reliable.

d - Limitation

Even though the acknowledgment scheme seems complete now, there is still a small probability that your program might lose some messages. Indeed, there is no acknowledgment for the *last* messages sent by a node before its termination.

For example, if the acknowledgment size is 20, and if the last message sent by a node is number 415, the 15 last messages will not be acknowledged, and if they are lost, you will have to run the simulation again.

There is no way to avoid this situation, except by controlling the way your CMMD program ends. And since the original program should not be modified, the situation is hopeless. However, this problem should be quite rare, and you can limit it by using a smaller acknowledgment size.

3 - How to choose the "acknowledgment size"

Let Time be the total amount of time needed for all the communications.

Time is the sum of the normal time to send the messages, the time to send again the messages not acknowledged, the time to send the acknowledgment messages and the time waiting for an acknowledgment.

$$Time = N * t + N * prob_loss * ACK * t + \frac{N}{ACK} * (t + wait)$$

where prob_loss is the probability to lose a message, ACK is the acknowledgment size, N the number of messages sent and "wait" the average time waiting for an acknowledgment.

If we consider the time of waiting negligible when compared to time to send a message, then Time is proportional to (1+ACK*prob_loss+(1/ACK)). We want to minimize this expression by choosing the right ACK. The derivation gives:

$$prob_loss - \frac{1}{ACK^2} = 0$$

which finally gives:

$$ACK = \sqrt{\frac{1}{prob_loss}}$$

with prob_loss=1%, we get ACK=10.

with prob_loss=0.2%, ACK=22.

It is difficult to evaluate the probability of the loss of a message. Nevertheless, we may consider that the good values of ACK are between 10 and 30. The experience allows me to say that 20 is a good value.

4 - Implementation of an ACK message

Acknowledgment messages are very simple messages. They contain only the number of the sender, of the receiver and the number of the message. Here is the exact implementation in the simulation:

```
class ack_message {

  public:
   int dest;
   int number;
   int source;

   // Constructors
}
```

This type of message is very simple, but very useful too !

IV - Active messages

1 - Normal active message

a - Composition

CMMD active messages are composed of five words (see A-II-4): one is the address of the handler function, and the four others are the arguments used by the handler. Nevertheless, the active messages of the simulation are longer, for several reasons:

- The number of arguments used by the handler is variable: four is the maximum, but it is possible to call a handler without argument. This is why the simulation has to know the number of arguments, and it is part of the active message.
- The acknowledgment scheme needs some data about each message. The active message must contain the number of the sender, of the receiver, and the number of the message itself.

However, it is important to handle short messages, because they are less likely to be lost or truncated. This is the reason why the address of the handler is implemented as a union:

it might be a handler with four, three, two , one or no argument, but it just uses one word in the message.

b - Implementation

Here is the C++ implementation of an active message

```
class active_message {
 public:
  int nb_arg;         // Number of arguments for handler
  int source;         // Sending node
  int dest;           // Destination node
  int number;         // Number of the message (for ACK)
  union {
    void (*handler4)(int,int,int,int);
    void (*handler3)(int,int,int);
    void (*handler2)(int,int);
    void (*handler1)(int);
    void (*handler0)();
    void (*handler)(char *);
  };
  int data[4];

 // Constructors

}
```

2 - Long active message

a - Presentation

Performance is a very important issue when simulating a CM-5. Indeed, the only reason why someone would buy such a powerful - and expensive - machine, is that it is the only way to achieve very high performance. Now, if the simulation is much slower, there is no point in using such a program. That is the reason why everything has been done in order to improve the simulation speed and performances.

One of the main limitations when using the CMAML library is that an active message only contains four words of data. When one needs to transmit more data, one has to cut the data in pieces, send several messages, and put the data back together. No need to say that it is painful (it requires some work !), and slows the program.

When working with UNIX, sending five messages that are five words long (twenty words of data) is obviously slower than sending one message which is twenty-one words long.

This is why the simulation has been designed to send active messages with an arbitrary long data part.

This has been added in order to provide a real speed-up in the simulation. Of course, to get this speed-up, you have to modify slightly the original CMMD program. Furthermore, every program is not likely to be improved: some only need the four words of data. But when a program needs to send longer data, the improvement is very noticeable.

b - Implementation

The new class is a sub-class of the normal active message class. It needs indeed the same data: sender, receiver, number, handler, and only one more: long_data.

Here is the actual C++ implementation:

```
class lg_active_message : public active_message {
  public:
     char long_data[DATA_SIZE];
  // Constructors
}
```

The new "CMMD" functions that use long messages are defined in **cmmd.h**:

```
// long argument handler
void CMAML_request(int,void (*)(char *),char *);
void CMAML_rpc(int,void (*)(char *),char *);
void CMAML_reply(int,void (*)(char *),char *);
```

The data are accessed through a pointer to a character.

c - Possible improvement

The "long_data" size is fixed when the program is compiled. Thus, the simulation might crash when the user wants to use a longer message (in fact, it will warn the user before). The idea is then either to recompile the program with a higher DATA_SIZE value, or to provide a function that would cut the message in several pieces, send it, and then put the pieces back together. This is not implemented in the current version of the simulation. However, this routine is pre-written. It has not been tested, and should not work yet, but it gives a good idea of how this could be done. See the request() function that uses a long argument handler for more details (file **node.cc**).

V - The launcher

1 - Presentation

The **go** program is the only executable which is part of the package. It is used to launch the node processes on each hosts, but it also has other functions. A preliminary presentation of this program can be found in I-5.

In fact, the **go** program has several features:

- It launches the processes on the hosts
- It allows the synchronization between nodes
- It detects the end of the program
- It provides several options

2 - Launching processes

This is the primary use of the **go** program. When a CMMD program has been successfully compiled with the library, it is still difficult to use: you have one executable, and you want it to run on several machines, with the different processes interacting with each other. This is how **go** does it:

a - Command-line arguments

The first thing to do is to store the command-line arguments which are to be given to each node. This is done automatically by **go.**

b - Reading the data from file "HOSTS"

This file contains all the data needed to launch the processes.

Here is an example of HOSTS file: (cf. I-4)

```
# Host      port    Command  Login   Name and  Location of binary file
hyde        12365   remsh    guerin  main      /users/guerin/project/example
chryse      10248   remsh    guerin  main      /auto/users/guerin/project/example
```

With this file, **go** will compose the actual commands that will launch the processes.

c - Launching the processes

For each line in the file "HOSTS", **go** will duplicate its process (using fork()), and will execute the launching command on the child process:

```
// fork process and launch program on child
if ((process[size]=fork())==0)   // The child process
        execl("/bin/sh", "sh", "-c",buf, 0);
```

The two launching commands corresponding to the "HOSTS" file given and invoked by the command "go 1 2 3 4" on the host "tenedos" are: (on hp workstations)

```
remsh hyde -l guerin 'cd /users/guerin/project/example; ./main 1 2 3 4 0 tenedos 7389 0 0'
remsh chryse -l guerin 'cd /auto/users/guerin/project/example; ./main 1 2 3 4 1 tenedos 7389 0 0'
```

The format of this command is as follow:

launch-command 'cd work_dir; ./program_name options node_number host_of_go port#_of _go verbose_type acknowledgment_size'

The meaning of the two last options will be explained later (in 5-).

When running **go** on SUN, "rsh" is used instead of "remsh". The launch_command is then "rsh -l guerin hyde ...".

3 - Synchronization

Once all processes are launched, **go** is still useful. It knows the addresses of all the sockets used by the nodes (since they are in the HOSTS file), and the nodes also know its address, which was given in the command-line arguments (host_of_go and port#_of go). This allows **go** to communicate with the nodes.

It is used when the nodes want to synchronize (command CMMD_sync_with_ node()). This command is a barrier synchronization where all nodes wait until all other nodes execute the function. Its implementation in the node will be described later (VI-3), but it seems obvious that the only way to implement this is to have a process that acts like the Partition Manager of the CM-5. And **go** is indeed an exact analog of the Partition Manager, and a synchronization is simple to implement.

go just waits for messages (blocking mode). When a message is received, and when this message is indeed a synchronization message, then a counter in incremented. When the counter reaches the size of the partition (i.e. the number of nodes running), then **go** sends a signal to every node, which means that all nodes are indeed executing the barrier synchronization.

This is the actual C++ implementation:

```
for(;;) {
    memset(mes,0,DIALOGUE_SIZE);
    // Wait for message, and read it
    n=recvfrom(launch_socket,mes,DIALOGUE_SIZE,0,&adr_temp,&length);

    // Test if sync_with_node message
    if (mes[0]=='@') { // sync_with_node
            counter++;
            if (counter==size) { // all nodes are waiting
            // reset counter
            counter=0;
            // send signal to all nodes
            for (i=0;i<size;i++) {
            if (sendto(launch_socket,&answer,1,0,&node_address[i],
                        sizeof(struct sockaddr_in))==-1) {
              cerr << "sendto\n";
              exit(2);
               }
            }
            if (vb) cout << "Launcher: Sync_with_nodes() succeded\n";
      }
    }
}
```

4 - Detection of the end of the program

How do you know when a program is terminated ? Usually, it will print a message saying that the computation has been completed. Nevertheless, when the program prints all its results in a file, you can not easily say when it is finished.

A small routine has been implemented to detect the end of the program.

As it was said before, each node process is launched by a child process of **go**. This program is therefore able to detect the termination of one of its child. This is achieved by catching the SIGCHLD signal, which is sent each time a child process ends. You can then count the number of child processes finished, and when this number equals the partition size, then the program is terminated.

Here is the implementation:

```
// ------------------------------
// Handler for SIGCHLD signal
// -----------------------------

void node_finished(int)
```

```
{
(void) signal(SIGCHLD,SIG_IGN);                     // Empty queue
(void) signal(SIGCHLD, node_finished);  // Ready to catch next signal
nb_finished++;
if (nb_finished==size) exit(0);
}

// ----------
// - main() -
// ----------

main(int argc,char **argv)
{

(void) signal(SIGCHLD, node_finished);
// When the signal SIGCHLD is caught,
// the function node_finished() will be called
...
```

This routine will not work properly if two signals are received exactly at the same time, which, though unlikely, might happen.

5 - Options

the **go** program offers several options that can be accessed through the command-line arguments.

- "go -h" will print the various options available:

usage: go [-port nnnn] [-v 1/2] [-ack nn] [-tofile] [node_arguments]

- "go -port 1234" will force the program to use port 1234 instead of the default port (7389). This is useful when the default port is already used. This explains why the port number is specified in the node command-line arguments.

- "go -v 1" and "go -v 2" are the verbose options. If you don't use this option, the program will execute exactly as it would on a CM-5. With the "-v 1" option, you will be given some information: the actual commands executed by **go** (remsh...), you will know when a sync_with_nodes() has been successfully executed, and you will be given the number of the messages that are sent several times by the ACK scheme. "go -v 2" is very verbose. It will give the same information as the "-v 1" option, plus it will print the number of every message sent, every message received, every acknowledgment etc. These options were primarily used when programming and debugging the library, and are probably useless to the user.

- "go -ack nn" is a very important option. This is the only way to activate the acknowledgment scheme. "nn" is the acknowledgment size, as described before. "go -ack 0" will run the simulation with the acknowledgment scheme disabled (same as "go" alone). "go -ack 1" will request an acknowledgment for every message sent. This will noticeably slow down the simulation. "go -ack 20" seems to be the best compromise for speed and reliability (acknowledgment every 20 messages). Be careful though, the acknowledgment size is limited! See **node.h** to know the actual limit (probably 30).

- "go -tofile" is useful when the output of the nodes is considerable, or when you need the output separated for each node. With this option enabled, the output of each node will be redirected into a file called "resultN" where "N" is the number of the node. This can be very useful when debugging a CMMD program.

VI - Initialization of the node

1 - Modification of the CMMD program

You will find the same text in I-6, but this is very important (and easy to forget).

Several parameters have to be given to each process, among which the indispensable logical address. Those parameters are given among the command-line arguments of the process. Therefore, the existing program has to be slightly modified.

A special function was created to handle the command-line arguments: CMMD_init_simulation(). In spite of its name, it is not a real CMMD function. It has been created for the simulation only. Not only does it handle the command-line arguments, but it also initiates the communications.

The only modification that you need to do in the existing program is to add these few lines just after the declaration of the main() function:

```
// to add if you are using the simulation; needs stdlib.h (atoi())
// begin
{
 // test number of arguments
 if (argc<6) {
   exit(2);
   }
```

```
  // Init with the 5 last arguments
  CMMD_init_simulation(atoi(*(argv+argc-5)),*(argv+argc-4),atoi(*(argv+argc-3)),
      atoi(*(argv+argc-2)),atoi(*(argv+argc-1)));

  // Ignore the 5 last arguments
  argc-=5;
}
// end
```

The “argc” and “argv” variables must be declared in the definition of main().

The CMMD program can now use its own command line arguments, since the others have been “forgotten”.

2 - CMMD_init_simulation()

The five arguments that have to be ignored by the CMMD program (cf. 1-), are those that were described in V-2-c: node_number, host_of_go, port#_of _go, verbose_type and acknowledgment_size.

The main() function of the CMMD program calls the CMMD_init_simulation() function with those five arguments: node_number, verbose_type and acknowledgment size are assigned to corresponding variables that will hold these values during the whole execution. The two other arguments are used to compute the address of the socket of the launcher.

Just like the program **go**, each node will read the file “HOSTS”. This file needs to be in the directories containing the executable on each host. Don’t forget to put this file in each directory! This is a source of very common mistakes.

The node will use the data contained in the file “HOSTS” to compute the address of every socket that will communicate with it. Each node has five sockets: one for each interface to the data network (request and reply), one for each corresponding acknowledgment channel, and finally one to communicate with **go** (for synchronization).

Once every address has been computed and every socket has been opened, every node execute the CMMD_sync_with_nodes() function, which will enable them to be synchronized: you don’t want a node to send a message to another one which has not even opened its own sockets.

VI - Functions implementation

1 - C++ implementation: the "node" class

In the C++ implementation of the simulation, each process launched uses an object of the "node" class. This class contains the data for the node: partition size, virtual address, the data for the communication: sockets, addresses of every other node, and the declaration of all the CMMD and CMAML functions.

In fact, an object of the class "node" is declared in the file **cmmd.cc**; and each call to a CMMD or CMAML function (declared in **cmmd.h**) is translated into a call to the corresponding "node" function (see the details in the listing of the files in the Appendix).

The "node" class is thus the central class of the simulation. It contains the actual implementation of the functions, whereas the other classes (active_message, ack_message, timer) are only tools manipulated by the node.

Here is the code of the "node" class:

```
class node {

  // the '[2]' means that the data is available for both interfaces:
  // 0 -> reply interface
  // 1 -> request interface

  int interrupt;            // interrupt on/off (1/0). Not used...
  int number;               // number of the node
  int size;                 // partition size
  int my_socket[2];         // socket on interfaces
  int socket_ack[2];        // socket for acknowledgments
  int socket_dialogue;      // socket allowing dialogue with launcher
  int vb;                   // verbose mode
  int is_ack[2][MAX_PARTITION_SIZE];  // Boolean, has ACK message been
                                      // received ?
  int counter_msg_received[2][MAX_PARTITION_SIZE];// Counters for
                                      // acknowledgments
  int counter_msg_sent[2][MAX_PARTITION_SIZE];   // counters of
                                      // messages sent
  int message_nb[2][MAX_PARTITION_SIZE]; // current message nb
  int message_nb_received[2][MAX_PARTITION_SIZE]; // last numbers
                                      // received

  // last messages sent :
  struct msg_box {
```

```
    int type;  // 0 for normal message, 1 for long
    active_message a_msg;
    lg_active_message lg_msg;
  }  message_sent[2][MAX_PARTITION_SIZE][MAX_ACK_SIZE];


  //addresses of all the other sockets
  struct sockaddr_in node_address[2][MAX_PARTITION_SIZE];
  struct sockaddr_in node_address_ack[2][MAX_PARTITION_SIZE];
  struct sockaddr_in launch_address;  // address of the launcher

  void test_ack(int,int);                       // Should I send the messages again ?
  void process_ack(ack_message *,int);          // process incoming ACK message
  void process_msg(active_message *,int);       // process incoming message
  void process_lg_msg(lg_active_message *,int);// process incoming long message
  void call_handler(active_message              // any explanation needed ?
  void call_lg_handler(lg_active_message *);    // any explanation needed ?
  void send_msg(active_message *,int);          // send active message
  void send_msg(lg_active_message *,int);       // send long active message
  void send_ack(ack_message *,int);             // send ACK message
  void send_ack(ack_message *,int,int);     // send ACK message, but modify number
  void poll_ack(int);                           // poll for ACK messages
  void resend_all_messages();               // resend all messages not acknowledged

 public:

  timer the_timer[64];
  int ack;          // acknowledgment mode: sending ACK every 'ack' messages

  // The following functions are called by the CMMD library

  void init(int,char*,int,int,int);
  int self_address() { return number; }
  int partition_size() { return size; }
  int disable_interrupts();
  int enable_interrupts();
  void poll(int);

  // long message handler
  void request(int,void (*)(char *),char *);
  void rpc(int,void (*)(char *),char *);
  void reply(int,void (*)(char *),char *);
  // 4 arguments handler
  void request(int,void (*)(int,int,int,int),int,int,int,int);
  void rpc(int,void (*)(int,int,int,int),int,int,int,int);
  void reply(int,void (*)(int,int,int,int),int,int,int,int);
(...)
  // 0 argument handler
  void request(int,void (*)());
```

```
    void rpc(int,void (*)());
    void reply(int,void (*)());

    void sync_with_nodes();
    int fset_io_mode(FILE *, int);
};
```

2 - Timers

The timers are implemented very simply (cf. Appendix for listing).

The timers of the simulation use the "time" library of UNIX (**time.h**). This is the only way to evaluate the "busy" time. But this method has a severe drawback: the timers are reset every 36 minutes, whereas the CM-5 timers are reset every 43 hours only. There is no way for the program to know when the timer is reset, and therefore, it is impossible to give an accurate number after 36 minutes. However, in some cases, it might be possible to know how many times the timer has been reset, by using an external clock.

The timers of the simulation only give the busy time. We have always:

```
CMMD_node_timer_elapsed() = CMMD_node_timer_busy()
CMMD_node_timer_idle() = 0
```

3 - CMMD_sync_with_nodes()

The global behavior of this function has already been described in V-3. Here is the "node part" of it.

When a node executes this function, it is expected to wait until every node also executes this function. The node simply sends an appropriate message to the launcher and waits for the answer (in blocking mode). When the launcher has received as many messages as the number of nodes in the partition, it sends the answer to every node.

4 - CMAML_poll()

The poll() function of the node class has been written so as to simulate CMAML_poll(), CMAML_poll_request() and CMAML_poll_reply(). This is achieved by using an integer parameter to the poll() function: 0 is CMAML_poll_reply(), 1 is CMAML_poll_request(), and 2 is CMAML_poll().

Here is the sequence of the instructions executed when polling:

- ♦ Test if the poll() function is being executed for the 2000th time in a row.
  - If it is true, a message might have been lost, so the node sends again all messages (function node::resend_all_messages()).

    - If not, the counter is incremented.

- The socket(s) is put in non-blocking mode.
- The program tries to receive a message on its socket. If no message is received, the program returns.
- If a message is received, then it is processed (function node::process_msg()).
    - The number of the message is compared to the number awaited. If it is an old message (already received), the corresponding acknowledgment is sent. If its number is to high, nothing is done: the previous message must have been lost. If it is a good message, then:
    - The counter is incremented.
    - If it has to send an acknowledgment, it is sent.
    - At last, the handler function is called.
- If there are other messages in the queue, they are also processed.

For more details, you can read the listing of **node.cc** in the Appendix.

5 - CMAML_request(), CMAML_reply(), CMAML_rpc()

In the actual version of the simulation, CMAML_request() and CMAML_rpc() are implemented in the same way. In fact, a call to CMAML_rpc() is simply replaced by a call to CMAML_request().

Here is the sequence of the instructions executed when sending an active message:

- The active message to be sent is created.
- The node tests if the previous message sent had to be acknowledged, and if so, if it has been acknowledged. If it has not been acknowledged then:
    - it polls 1000 times for an acknowledgment.
    - If no acknowledgment was received, then it sends again the whole sequence of messages that should have been acknowledged, and returns in the previous loop (waiting for an acknowledgment).
- The socket is put in blocking mode.
- The active message is sent.

- The message is stored, the counter incremented.
- The node polls for messages (only reply interface if it is a reply active message, or both if not).

For the exact implementation, see the listing of **node.cc** in the Appendix.

6 - Others

a - CMMD_self_address()

This function returns the virtual address of the node, which was initialized by CMMD_init_simulation().

b - CMMD_partition_size()

This function returns the size of the partition, which corresponds to the number of entries in the HOSTS file.

c - CMMD_disable_interrupt(), CMAML_disable_interrupt(), CMMD_fset_io_mode()

Those functions are provided for compatibility only. And they do... nothing at all.

VII - Limitations

1 - Homogeneous network

The simulation was designed to use many workstations in the same time, so as to provide good performances and speed. Unfortunately, it is not possible to use different types of workstations in the same time.

The reason to this limitation is very simple. An active message contains the address of the handler function to call. This address is local to the sending node, but if the receiver is the same type of station, running the same program, the address will indeed correspond to the handler function. However, if the platform is not the same, it is highly improbable that the address will correspond to anything, and the handler can not be called.

This is why you can not mix several types of workstations when using the simulation. You can use SUN Sparc stations, hp stations, but not both in the same time. However, you might be able to use slightly different stations such as hp 9000/750 and hp 9000/755, or SUN Sparc 4 and SUN Sparc 5.

2 - Number of nodes

Even if there is no explicit limitation in the number of nodes, you might get into trouble if you want to use too many nodes.

The limitation is caused by UNIX. Each time the program **go** launches a process, it has to fork, i.e. to duplicate its own process. The problem is that the number of processes running on a UNIX machine are limited. And after fifteen or twenty nodes launched, you might get the error message: “Can not fork: too many processes”.

However, this depends on the machine you use, and on the number of people who use it in the same time. You might be able to get many nodes if you run the program **go** on the appropriate machine.

C - PERFORMANCES

I - Benchmarks

1 - The "nt-tak" program

a - Presentation

This program has been provided by Steve Jenks. He is using it as a benchmark for his own research on nomadic threads.

b - Implementation

The "nt-tak" program is an implementation of a Lisp program (tak) using nomadic threads. This program takes three arguments and finally return one. Here is the original Lisp "tak":

```
( defun tak ( x y z)
    (if (>= y x)
      z
      (tak (tak (1 - x) y z)
          (tak (1 - y) z x)
          (tak (1 - z) x y))))
```

c - features

This program is a good benchmark program because:

- The number of messages exchanged between nodes is very large.
- The data that have to be exchanged between nodes is larger than the normal size of an active message. Therefore, it will be possible to use long active messages on the simulation.

- The time of completion decreases exponentially with the number of nodes, i.e. if N nodes finish in t seconds, then 2N nodes finish in t/2 seconds.

d - Results

Here are a few results of the "tak" program:

1st Arg	2nd Arg	3rd Arg	TAK result	Nb of activations
2	3	4	4	1
5	4	2	4	21
6	4	2	3	53
8	6	2	3	469
10	6	2	3	1733
12	6	2	3	4321
18	12	6	7	63609
20	12	6	7	155449
17	16	5	16	632965
25	20	10	20	6895965

2 - The "nt-pmm" program

a - Presentation

This program has also been provided by Steve Jenks, and it is also using nomadic threads. It is nevertheless quite different from "nt-tak".

b - Implementation

The "nt-pmm" program is a simple implementation of a matrix multiply program. It takes the matrix size as an argument, and only returns the time took by the execution of the program.

c - features

This program is also good benchmark program because:

- The number of messages exchanged between nodes is very large.
- It is quite different from the "nt-tak" program, and it helped me find some defaults in the acknowledgment scheme I would not have find without (see B-III-2-c).
- The data that have to be exchanged between nodes is larger than the normal size of an active message. Therefore, it will be possible to use long active messages on the simulation.

II - The platform issue

The simulation has been tested successfully on both SUN and hp workstations. The SUN version has been compiled with the GNU C++ Compiler (g++), whereas the hp version has been tested with both g++ and hp's C++ compiler (CC). Of course, performances are not the same on every platforms, and here is a quick comparison of the workstations tested.

The benchmark program is nt-tak, and the command was:

```
go -ack 20 18 12 6
```

which means that the arguments are 18 12 and 6, and that the acknowledgment size is 20. In every cases, only two nodes where used. The simulation is using long active messages.

Workstation	Time to complete
hp 9000/755	35 s
SUN Sparc 4	41 s - 57 s

The two values given for the SUN Sparc 4 have been obtained in two configurations. In the first case, the two stations were very close to each other, and no message has been lost. In the second case, the stations were far away, and many messages had to be sent several times.

This shows the importance of using close workstations with reliable communications. It might be a bad idea to add a workstation in the partition if it is far away from all the others.

III - Influence of the number of nodes

The following results have been obtained with "nt-tak" and the command: "go -ack 20 18 12 6". All nodes are SUN Sparc 4 workstations linked with ethernet, and "nt-tak" is using long active messages. The maximum number of nodes used is 12. This limitation comes from the fact that the process table of the machine which hosted the launcher was full. This limitation has been described in B-7-2.

1 - Time to complete

Number of nodes	Time
2	41
3	21
4	21
5	14.5
8	11.5
10	10
12	9.7

The following figure (fig. 3) is a simple representation of this table.

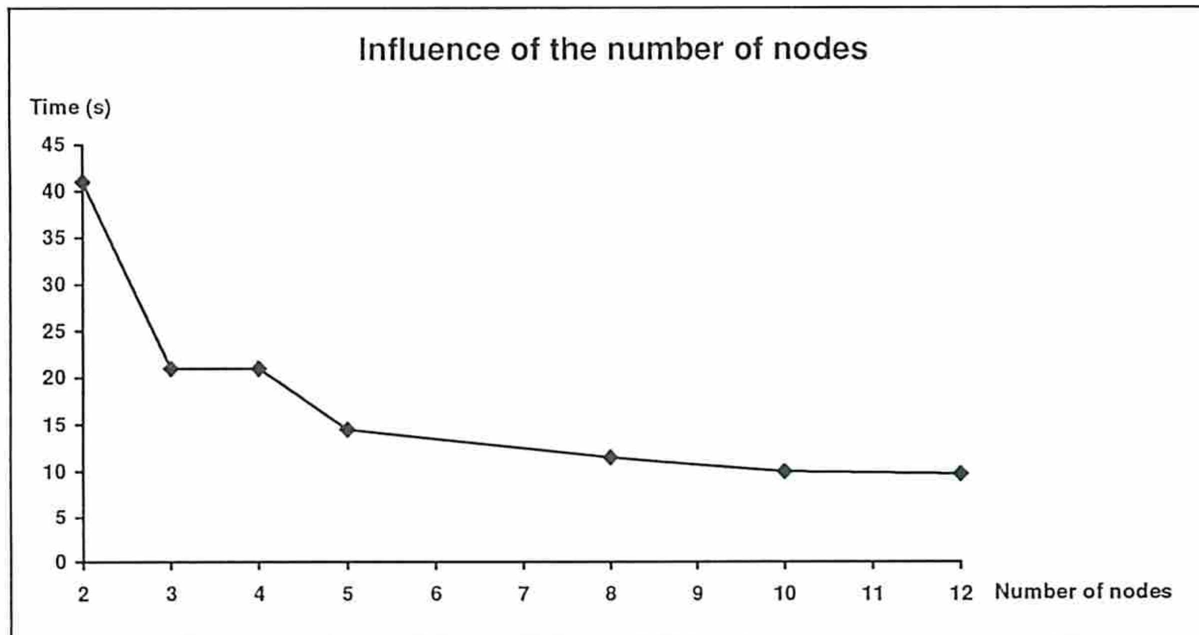

figure 3. Influence of the number of nodes on the time of completion

The figure obtained is a rough decreasing exponential, as expected. However, the value obtained with four nodes is quite different (or is it the value with three nodes ?). There is no improvement between a three and a four-node partition. This is probably due to the benchmark program. However, this should be verified on the CM-5.

2 - Overall efficiency

The following graph (fig. 4) shows the total amount of CPU time used to complete the computation. This was obtained by multiplying the time to complete by the number of nodes.

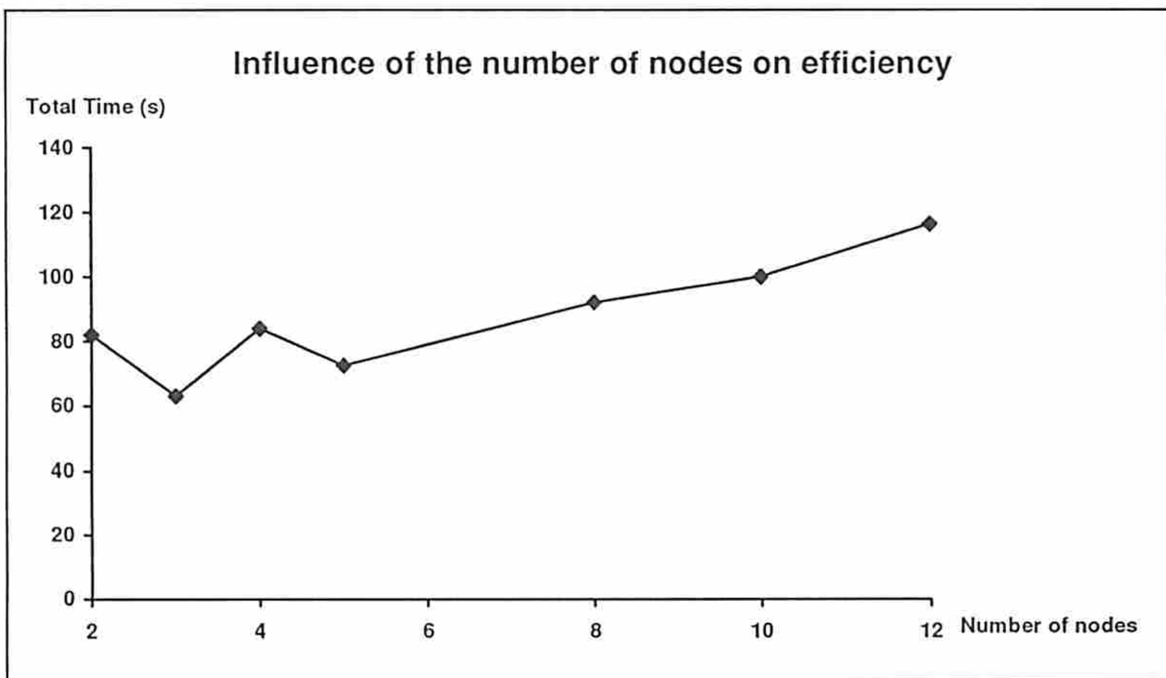

Figure 4. Total CPU time used to complete computation

Except for the anomaly at four nodes, the total CPU time seems to be growing steadily with the number of nodes. The probable cause is that the first few hosts were close, and very few messages were lost, but when you increase the number of hosts, you also increase the distance between them, and thus lose more messages.

It is also possible that, with so many messages going through it, the network is saturating, and more messages are lost.

Figure 5 shows the global speed-up obtained by increasing the number of nodes. Since the benchmark program does not give a good information when run with one node, the base value is obtained with two nodes.

The straight line shows the ideal speed-up (proportional to the number of nodes).

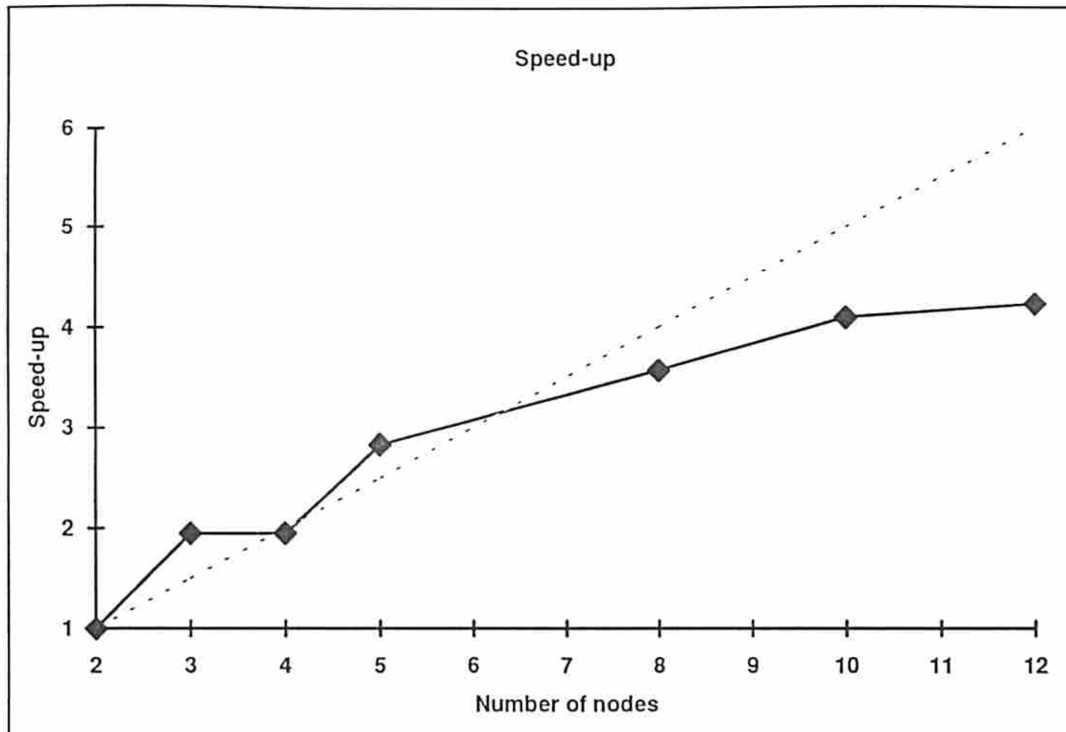

Figure 5. Speed-up obtained by increasing the number of nodes

Well, the results are obviously not perfect. However, it seems that the "nt-tak" program is not a very good benchmark to measure the influence of the number of nodes: the speed-up values obtained with three and five nodes are even better than the ideal ones.

It is also important to keep in mind that those benchmarks are based on heavy communication, and not on heavy computation. It might also explain the results that were obtained.

It would be interesting to compare this aspect of the simulation with the CM-5.

IV - Using long active messages

The gain obtained using long messages depends entirely on the application used. If only 4 data words are needed, using long messages won't improve the performances. However, if your program needs to exchange longer data between nodes, this feature can be very powerful.

The "nt-tak" program needs up to 12 data words per message (but sometimes less). Its performances are therefore improved by using long active messages.

The following times have been established in similar conditions: same hosts (two nodes) and same options (ACK size: 20):

Here is the corresponding speed-up:

Arguments	Normal active messages	Long active messages
go 18 12 6	79 s	35 s
go 17 16 5	789 s	329 s

The average speed-up is about 140% ! It is now obvious that this feature can be really useful, and I sometimes wonder why the CM-5 is limited to 4 words...

V - Comparison with the CM-5

These are the important results everybody is waiting for...

The following times have been obtained using the "nt-tak" program with the arguments 17 16 5 on two nodes. The simulation used an acknowledgment size of 20, and the two nodes are hp 9000/755.

Platform	Time	Comparison/CM-5
CM-5	123.27 s	1
hp 9000/755 Normal messages	789.03 s	6.4
hp 9000/755 Long messages	349.09 s	2.8

The simulation running on hp 9000/755 is therefore 6.4 times slower than the CM-5. This is not as bad as it seems, since the CM-5 has been optimized for inter-node communications, whereas the simulation is running on standard workstations. It is also important to notice that the benchmark program is based on frequent message exchanges (more than 1,000,000 in this configuration), which is where the CM-5 is the best. A program that needs more computation in the node itself would probably give better results.

The "long message" option is the only feature that can improve the performance of the simulation, compared to the CM-5. For the "nt-tak" program, the simulation runs only 2.8 times slower than the CM-5. This is a good result, since 2.8*N workstations are less expensive than a CM-5 with N nodes.

CONCLUSION

The main purpose of this project is to simplify the programming and debugging of a CM-5 program. It should be easier to debug programs on a workstation acting like a CM-5 than on the actual machine. The final version of the program is still supposed to run on the CM-5.

However, the simulation is "only" six times slower than the CM-5, and this number can be decreased to three or less by using long active messages. And, a CM-5 node is much more expensive than three workstations.

Furthermore, a CM-5 can only be used for extensive and huge computations, whereas a network of workstations is multi-purpose and much more flexible. For example, each computer can be used individually during the day, and simulate a CM-5 at night for intensive computations. It can even be done in the same time! Or some stations might be dedicated to the simulation whereas others are used differently...

However, a CM-5 might be the best choice when you need to make huge computations all day long, for example in a research laboratory. And the simulation also has some limitations on the total number of nodes. But it is probably safer to consider other options, like a network of workstations, before buying such an expensive machine.

BIBLIOGRAPHY

[1] CMMD reference Manual

Thinking Machines Corporation, Cambridge, Massachusetts

[2] UNIX System V Network Programming

Stephen A. Rago
Addison - Wesley Professional Computing Series

[3] The C++ Programming Language, Second Edition

Bjarne Stroustrup
Addison - Wesley Publishing Company

APPENDIX

Listing of the simulation:

- cmmd.h
- cmmd.cc
- timer.h
- timer.cc
- ack_msg.h
- active_msg.h
- active_msg.cc
- lg_act_msg.h
- lg_act_msg.cc
- node.h
- node.cc
- go.cc
- Makefile

Listing of the example:

- main.cc
- Makefile
- example of HOSTS file

```
/* cmmd.h - dummy header file to simulate the real cmmd.h
 *
 * History:
 *  4/22/95 - created by S. Jenks
 *  5/09/95 - modified by N. Guerin
 *  6/28/95 - last modification by N. Guerin
 */

#ifndef CMMD_H
#define CMMD_H

extern "C" {
#include <stdio.h>  // for the FILE type
}

void CMMD_init_simulation(int,char*,int,int,int);  // NOT a real CMMD function, on
+ ly used for simulation

// interrupts are not handled yet
void CMMD_disable_interrupts();
void CMAML_disable_interrupts();


#define CMMD_independent 0
//int CMMD_fset_io_mode(struct _IO_FILE *, int);
int CMMD_fset_io_mode(FILE *, int);  // not implemented

int CMMD_self_address();
int CMMD_partition_size();

int CMMD_node_timer_clear(int);
int CMMD_node_timer_start(int);
int CMMD_node_timer_stop(int);
// WARNING: the timer resets after 36 minutes only (43 hours for real CM-5)
double CMMD_node_timer_elapsed(int);
double CMMD_node_timer_busy(int); // for compatibility only = elapsed()
double CMMD_node_timer_idle(int); // for compatibility only = 0

void CMMD_sync_with_nodes();

void CMAML_poll();
void CMAML_request_poll();
void CMAML_reply_poll();

// CMAML functions which send messages (request, reply and RPC)
// WARNING: for now, RPC = request
// long argument handler
void CMAML_request(int,void (*)(char *),char *);
void CMAML_rpc(int,void (*)(char *),char *);
void CMAML_reply(int,void (*)(char *),char *);
// 4 arguments handler
void CMAML_request(int,void (*)(int,int,int,int),int,int,int,int);
void CMAML_rpc(int,void (*)(int,int,int,int),int,int,int,int);
void CMAML_reply(int,void (*)(int,int,int,int),int,int,int,int);
// 3 arguments handler
void CMAML_request(int,void (*)(int,int,int),int,int,int);
void CMAML_rpc(int,void (*)(int,int,int),int,int,int);
void CMAML_reply(int,void (*)(int,int,int),int,int,int);
// 2 arguments handler
void CMAML_request(int,void (*)(int,int),int,int);
void CMAML_rpc(int,void (*)(int,int),int,int);
void CMAML_reply(int,void (*)(int,int),int,int);
// 1 argument handler
void CMAML_request(int,void (*)(int),int);
void CMAML_rpc(int,void (*)(int),int);
void CMAML_reply(int,void (*)(int),int);
// 0 argument handler
void CMAML_request(int,void (*)());
void CMAML_rpc(int,void (*)());
void CMAML_reply(int,void (*)());

#endif /* CMMD_H */
```

```
/* cmmd.cc - Implementation of the CMMD and CMAML fuctions using the node class
 *
 * History:
 *   05/09/95 - Created by N. Guerin
 *   06/28/95 - Last modification by N. Guerin
 */

#include "cmmd.h"
#include "node.h"


static node the_node;

void CMMD_init_simulation(int nb,char *hostname,int port,int verbose,int ackno) {
        // NOT a real CMMD function, only used for simulation
        the_node.init(nb,hostname,port,verbose,ackno); }

void CMMD_disable_interrupts() { the_node.disable_interrupts(); }

void CMAML_disable_interrupts() { the_node.disable_interrupts(); }


//int CMMD_fset_io_mode(struct _IO_FILE *stream, int io_mode)
int CMMD_fset_io_mode(FILE *stream, int io_mode)
        { return the_node.fset_io_mode(stream,io_mode); }

int CMMD_self_address() { return the_node.self_address(); }
int CMMD_partition_size() { return the_node.partition_size(); }

// timer functions
int CMMD_node_timer_clear(int nb) { return the_node.the_timer[nb].clear(); }
int CMMD_node_timer_start(int nb) { return the_node.the_timer[nb].start(); }
int CMMD_node_timer_stop(int nb) { return the_node.the_timer[nb].stop(); }
double CMMD_node_timer_elapsed(int nb) { return the_node.the_timer[nb].elapsed();
+ }
// for compatibility only (busy=elapsed)
double CMMD_node_timer_busy(int nb) { return the_node.the_timer[nb].elapsed(); }
// for compatibility only (idle=0)
double CMMD_node_timer_idle(int) { return (double)0; }


// Communication functions

void CMMD_sync_with_nodes(void) { the_node.sync_with_nodes(); }

// poll for messages
void CMAML_poll() {the_node.poll(2); }
void CMAML_request_poll() { the_node.poll(1); }
void CMAML_reply_poll() { the_node.poll(0); }

// send request, reply and RPC messages

// long argument handler
void CMAML_request(int dest_node,void (*handler)(char *),char *data1) {
   the_node.request(dest_node,handler,data1);
   }
void CMAML_rpc(int dest_node,void (*handler)(char *),char *data1) {
   the_node.rpc(dest_node,handler,data1);
   }
void CMAML_reply(int dest_node,void (*handler)(char *),char *data1) {
   the_node.reply(dest_node,handler,data1);
   }

// 4 arguments handler
void CMAML_request(int dest_node,void (*handler)(int,int,int,int),int data1,int da
+ ta2,int data3,int data4)
   { the_node.request(dest_node,handler,data1,data2,data3,data4);    }
void CMAML_rpc(int dest_node,void (*handler)(int,int,int,int),int data1,int data2,
+ int data3,int data4)
   { the_node.rpc(dest_node,handler,data1,data2,data3,data4); }
void CMAML_reply(int dest_node,void (*handler)(int,int,int,int),int data1,int data
+ 2,int data3,int data4)
   { the_node.reply(dest_node,handler,data1,data2,data3,data4); }

// 3 arguments handler
void CMAML_request(int dest_node,void (*handler)(int,int,int),int data1,int data2,
+ int data3)
   { the_node.request(dest_node,handler,data1,data2,data3); }
void CMAML_rpc(int dest_node,void (*handler)(int,int,int),int data1,int data2,int
+ data3)
   { the_node.rpc(dest_node,handler,data1,data2,data3); }
void CMAML_reply(int dest_node,void (*handler)(int,int,int),int data1,int data2,in
+ t data3)
   { the_node.reply(dest_node,handler,data1,data2,data3); }

// 2 arguments handler
void CMAML_request(int dest_node,void (*handler)(int,int),int data1,int data2)
   { the_node.request(dest_node,handler,data1,data2); }
void CMAML_rpc(int dest_node,void (*handler)(int,int),int data1,int data2)
   { the_node.rpc(dest_node,handler,data1,data2); }
void CMAML_reply(int dest_node,void (*handler)(int,int),int data1,int data2)
   { the_node.reply(dest_node,handler,data1,data2); }

// 1 argument handler
void CMAML_request(int dest_node,void (*handler)(int),int data1)
   { the_node.request(dest_node,handler,data1); }
void CMAML_rpc(int dest_node,void (*handler)(int),int data1)
   { the_node.rpc(dest_node,handler,data1); }
void CMAML_reply(int dest_node,void (*handler)(int),int data1)
   { the_node.reply(dest_node,handler,data1); }

// 0 argument handler
void CMAML_request(int dest_node,void (*handler)())
   { the_node.request(dest_node,handler); }
void CMAML_rpc(int dest_node,void (*handler)())
   { the_node.rpc(dest_node,handler); }
void CMAML_reply(int dest_node,void (*handler)())
   { the_node.reply(dest_node,handler); }
```

```
/* timer.h - Implementation of the CMMD timers
 *
 * History:
 *   05/09/95 - Created by N. Guerin
 *   06/01/95 - Last modification by N. Guerin
 */

extern "C" {
#include <time.h>
}

#ifndef CLOCKS_PER_SEC
#  define CLOCKS_PER_SEC 1000000  // is it OK for all systems ?
#endif

class timer {
   clock_t begin;      // start time
   clock_t end;        // stop time
   double counter;     // private counter
 public:
   int clear();       // reset timer
   int start();
   int stop();
   double elapsed(); // in seconds
};
```

```
/* timer.cc
 *
 * History:
 *   05/09/95 - Created by N.Guerin
 *
 */

#include "timer.h"

// ATTENTION :
// timer resets after 36 minutes
// CMMD timer : 43 hours !

int timer::clear() {       // reset timer
  counter=0;
  return 0;
}

int timer::start() {
  begin=clock();
  return 0;
}

int timer::stop() {
  end=clock();
  counter+=double(end-begin)/CLOCKS_PER_SEC;
  return 0;
}

double timer::elapsed() { // in seconds
return counter;
}
```

```
/* ack_msg.h
 *
 * History:
 *   06/14/95 - Created by N.Guerin
 *   06/28/95 - Last modification by N.Guerin
 */

class ack_message {

  public:
   int dest;
   int number;
   int source;

   // Constructors
   ack_message(int src,int des,int num){
      source=src;
      dest=des;
      number=num;
   };

   ack_message(){
      source=0;
      dest=0;
      number=0;
   };

 };
```

```
/* active_msg.h
 *
 * History:
 *   05/17/95 - Created by N.Guerin
 *   06/28/95 - Last modification by N.Guerin
 */

#include "ack_msg.h"

class active_message {
  public:
   int nb_arg; // Number of arguments for handler
   int source; // Sending node
   int dest;   // Destination node
   int number; // Number of the message (for ACK)
   union {
      void (*handler4)(int,int,int,int);
      void (*handler3)(int,int,int);
      void (*handler2)(int,int);
      void (*handler1)(int);
      void (*handler0)();
      void (*handler)(char *);
   };
   int data[4];

   active_message(int,int,int,void (*)(int,int,int,int),int,int,int,int);
   active_message(int,int,int,void (*)(int,int,int),int,int,int);
   active_message(int,int,int,void (*)(int,int),int,int);
   active_message(int,int,int,void (*)(int),int);
   active_message(int,int,int,void (*)());
   active_message();
   void ack(ack_message *);
   int msg_source() {return source;}
   int msg_dest()   {return dest;}
   int msg_number() {return number;}
};
```

```
/* active_msg.cc
 *
 * History:
 *    05/17/95 - Created by N.Guerin
 *    06/28/95 - Last modification by N.Guerin
 */

#include "active_msg.h"


void active_message::ack(ack_message *ack_msg){
   ack_msg->source=source;
   ack_msg->dest=dest;
   ack_msg->number=number;
}

// Constructors

active_message::active_message(int src,int des,int num,void (*hand)(int,int,int,in
+ t),int data1,int data2,                          int data3,int data4){
    data[0]=data1;
    data[1]=data2;
    data[2]=data3;
    data[3]=data4;
    nb_arg=4;
    handler4=hand;
    source=src;
    dest=des;
    number=num;
}

active_message::active_message(int src,int des,int num,void (*hand)(int,int,int),i
+ nt data1,int data2,                          int data3){
    data[0]=data1;
    data[1]=data2;
    data[2]=data3;
    data[3]=0;
    nb_arg=3;
    handler3=hand;
    source=src;
    dest=des;
    number=num;
}

active_message::active_message(int src,int des,int num,void (*hand)(int,int),int d
+ ata1,int data2){
    data[0]=data1;
    data[1]=data2;
    data[2]=0;
    data[3]=0;
    nb_arg=2;
    handler2=hand;
    source=src;
    dest=des;
    number=num;
}

active_message::active_message(int src,int des,int num,void (*hand)(int),int data1
+ ){
    data[0]=data1;
    data[1]=0;
    data[2]=0;
    data[3]=0;
    nb_arg=1;
    source=src;
    dest=des;
    number=num;
    handler1=hand;
}

active_message::active_message(int src,int des,int num,void (*hand)()){
    data[0]=0;
    data[1]=0;
    data[2]=0;
    data[3]=0;
    nb_arg=0;
    handler0=hand;
    source=src;
    dest=des;
    number=num;
}

active_message::active_message(){
    data[0]=0;
    data[1]=0;
    data[2]=0;
    data[3]=0;
    nb_arg=4;
    handler4=0;

    source=0;
    dest=0;
    number=0;
}
```

```
/* lg_act_msg.h
 *
 * History:
 *   06/26/95 - Created by N.Guerin
 *   06/28/95 - Last modification by N.Guerin
 */

#include <string.h>
#include "active_msg.h"

#define  DATA_SIZE      57  // OK for TAK program


class lg_active_message : public active_message {

  public:
   char long_data[DATA_SIZE];
   char to_be_contd; // Part of a longer msg (long arg)

   lg_active_message(int,int,int,char,void (*)(char *),char *);
   lg_active_message(int,int,int,char,void (*)(char *));
   lg_active_message();
   int msg_source() {return source;}
   int msg_dest() {return dest;}
   int msg_number() {return number;}
};
```

```
/* lg_act_msg.cc
 *
 * History:
 *   06/26/95 - Created by N.Guerin
 *   06/28/95 - Last modification by N.Guerin
 */

#include "lg_act_msg.h"


// Constructors

lg_active_message::lg_active_message(int src,int des,int num,char contd,void (*han
+ d)(char *),char *data1){
   memcpy(long_data,data1,DATA_SIZE);
   nb_arg=-1;
   source=src;
   dest=des;
   number=num;
   handler=hand;
   to_be_contd=contd;
}

lg_active_message::lg_active_message(int src,int des,int num,char contd,void (*han
+ d)(char *)){
   nb_arg=-1;
   source=src;
   dest=des;
   number=num;
   handler=hand;
   to_be_contd=contd;
}

lg_active_message::lg_active_message(){

   nb_arg=-1;
   handler=0;
   to_be_contd=0;
   source=0;
   dest=0;
   number=0;
}
```

```
/* node.h
 *
 * History:
 *   05/09/95 - Created by N. Guerin
 *   08/22/95 - Last modification by N. Guerin
 */

#include "timer.h"
#include "lg_act_msg.h"

#include <fstream.h>
#include <libc.h>

extern "C" {
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <signal.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <netdb.h>
#include <unistd.h>
#include <sys/socket.h>
#ifdef _SUN
  #include <sys/filio.h>
#endif
}

#define STRING_SIZE     255
#define DIALOGUE_SIZE   1
#define MAX_PARTITION_SIZE      16
#define MAX_ACK_SIZE    30

#define AF      AF_INET


class node {

   // the '[2]' means that the data is available for both interfaces:
   // 0 -> reply interface
   // 1 -> request interface

   int interrupt;              // interrupt on/off (1/0). Not used...
   int number;                 // number of the node
   int size;                   // partition size
   int my_socket[2];           // socket on interfaces
   int socket_ack[2];          // socket for acknowledgments
   int socket_dialogue;        // socket allowing dialogue with launcher
   int vb;                     // verbose mode
   int is_ack[2][MAX_PARTITION_SIZE]; // Boolean, has ACK message been received ?
   int counter_msg_received[2][MAX_PARTITION_SIZE];// Counters for ack
   int counter_msg_sent[2][MAX_PARTITION_SIZE];     // counters of messages sent
   int message_nb[2][MAX_PARTITION_SIZE];           // current message nb
   int message_nb_received[2][MAX_PARTITION_SIZE]; // last numbers received

   // last messages sent
   struct msg_box {
      int type;  // 0 for normal message, 1 for long
      active_message a_msg;
      lg_active_message lg_msg;
   }  message_sent[2][MAX_PARTITION_SIZE][MAX_ACK_SIZE];


   //addresses of all the other sockets
   struct sockaddr_in node_address[2][MAX_PARTITION_SIZE];
   struct sockaddr_in node_address_ack[2][MAX_PARTITION_SIZE];
   struct sockaddr_in launch_address;  // address of the launcher

   void test_ack(int,int);                // Should I send the messages again ?
   void process_ack(ack_message *,int);          // process incoming ACK message
   void process_msg(active_message *,int);       // process incoming message
   void process_lg_msg(lg_active_message *,int);// process incoming long message
   void call_handler(active_message *);          // any explanation needed ?
   void call_lg_handler(lg_active_message *);    // any explanation needed ?
   void send_msg(active_message *,int);          // send active message
   void send_msg(lg_active_message *,int);       // send long active message
   void send_ack(ack_message *,int);             // send ACK message
   void send_ack(ack_message *,int,int);// send ACK message, but modify number...
   void poll_ack(int);                    // poll for ACK messages
   void resend_all_messages();            // resend all messages not acknowledged

public:

   timer the_timer[64];
   int ack;            // acknowledgment mode: sending ACK every 'ack' messages

   // The following functions are called by the CMMD library

   void init(int,char*,int,int,int);
   int self_address() { return number; }
   int partition_size() { return size; }
   int disable_interrupts();
   int enable_interrupts();
   void poll(int);

   // long message handler
   void request(int,void (*)(char *),char *);
   void rpc(int,void (*)(char *),char *);
   void reply(int,void (*)(char *),char *);
   // 4 arguments handler
   void request(int,void (*)(int,int,int,int),int,int,int,int);
   void rpc(int,void (*)(int,int,int,int),int,int,int,int);
   void reply(int,void (*)(int,int,int,int),int,int,int,int);
   // 3 arguments handler
   void request(int,void (*)(int,int,int),int,int,int);
   void rpc(int,void (*)(int,int,int),int,int,int);
   void reply(int,void (*)(int,int,int),int,int,int);
   // 2 arguments handler
   void request(int,void (*)(int,int),int,int);
   void rpc(int,void (*)(int,int),int,int);
   void reply(int,void (*)(int,int),int,int);
   // 1 argument handler
   void request(int,void (*)(int),int);
   void rpc(int,void (*)(int),int);
   void reply(int,void (*)(int),int);
   // 0 argument handler
   void request(int,void (*)());
   void rpc(int,void (*)());
   void reply(int,void (*)());

   void sync_with_nodes();
   int fset_io_mode(FILE *, int);
};
```

```
/* node.cc - Main body of the CMMD simulation
 *
 * History:
 *   05/09/95 - Created by N. Guerin
 *   08/22/95 - Last mofification by N. Guerin
 */

#include "node.h"

static int waiting=0;    // see poll() function
static int resending=0;  // see poll() function

// ---------------------------------------------------------
// - create_socket() - General function that creates a socket -
// -                   on a specified port                   -
// ---------------------------------------------------------

static int create_socket(int *port,int type)
{
        int desc;                      /* descriptor of the new socket */
        struct sockaddr_in name;       /* address of the socket */
        int length;                    /* Length of address */

        /* Creation of the socket */
        if ((desc=socket(AF,type,0))==-1){
                cerr << "Cannot create socket\n";
                return(-1);
                }
        memset((char *)&name,0,sizeof(name));
        name.sin_port=*port;
        name.sin_addr.s_addr=INADDR_ANY;
        name.sin_family=AF;
        if(bind(desc,&name,sizeof(name))){
                fprintf(stderr,"Node: Can't bind socket\n");
                return(-1);}
        length=sizeof(name);
        if(getsockname(desc,&name,&length)){
                perror("Cannot get socket name\n");
                return(-1);}
        *port=ntohs(name.sin_port);
        return(desc);
}


// ---------------------------------------------------
// - setsocknoblock() - Set socket in non-blocking mode -
// ---------------------------------------------------

static inline int setsocknoblock(int fd) { //fd = file descriptor
   int flag = 1;
   return(ioctl(fd,FIONBIO,&flag));
}


// -----------------------------------------------
// - setsockblock() - Set socket in blocking mode -
// -----------------------------------------------

static inline int setsockblock(int fd) { //fd = file descriptor
   int flag = 0;
   return(ioctl(fd,FIONBIO,&flag));
}
```

```
// ------------------------------------------------
// - init() - initialization of the communications -
// ------------------------------------------------

void node::init(int nb,char *launch_host,int launch_port,int verbose,int ackno) {

  int dialogue_port;
  char comment;
  char buf[STRING_SIZE];
  char host_buf[STRING_SIZE];
  int  my_port[2];
  int  port_ack[2];
  struct hostent *hp;
  int i,j,req;

  ack=ackno;
  vb=verbose;

  // get node number
  number=nb;


  if (ack) for (i=0; i<MAX_PARTITION_SIZE;i++) {
   for(req=0;req<2;req++) {
     is_ack[req][i]=1;
     counter_msg_received[req][i]=0;
     counter_msg_sent[req][i]=0;
     message_nb[req][i]=0;
     message_nb_received[req][i]=0;
     for (j=0;j<ack;j++) {
        message_sent[req][i][j].type=0;
        memset(&message_sent[req][i][j].a_msg,0,sizeof(active_message));
        memset(&message_sent[req][i][j].lg_msg,0,sizeof(lg_active_message));
        }
     }
   }

  // - change stdout to stderr -
  // messages will be printed on terminal
  // First, close stdout
  close(1);
  // and then, duplicate stderr (new stdout)
  dup(2);

  // -----------------------------------------
  // - Read datas: partition_size, addresses... -
  // -----------------------------------------


  // Open file HOSTS
  ifstream from("HOSTS");
  if (!from) {
     cerr << "Cannot open file HOSTS\n";
     exit(1);
     }

  // Read the file

  size=0;

  while (from.getline(buf,STRING_SIZE)) {

     // forget it if it is a comment
     comment=buf[0];
     if ((comment=='#')||(comment==' ')||(comment=='\n')||(comment==0))
         continue;
     if (!sscanf(buf,"%s %d",host_buf,&my_port[0])) continue;
```

```
    port_ack[0]=my_port[0]+1;
    my_port[1]=my_port[0]+2;
    port_ack[1]=my_port[0]+3;
    dialogue_port=my_port[0]+4;
    if (size==nb) {   // My data
       // Open dialogue socket
       if ((socket_dialogue=create_socket(&dialogue_port,SOCK_DGRAM))==-1){
          cout << "Node"<<number<<": Cannot create dialogue socket\n";
          exit(1);
          }
       if (vb) cout << "Node" << number <<": Socket "<< socket_dialogue << " open
+ ed on port "<< dialogue_port<<'\n';
       }

    // Compute addresses

    if((hp = gethostbyname (host_buf)) == 0)
          {
          cerr << "Cannot find address\n";
          exit(1);
          }
    for (req=0;req<2;req++) {
       memset (&node_address[req][size],0, sizeof (struct sockaddr_in));
       node_address[req][size].sin_family = AF;
       node_address[req][size].sin_port = htons (my_port[req]);
       memcpy(&node_address[req][size].sin_addr,hp->h_addr_list[0],hp->h_length);
    }

    if (ack) {
     for (req=0;req<2;req++) {
       memset (&node_address_ack[req][size],0, sizeof (struct sockaddr_in));
       node_address_ack[req][size].sin_family = AF;
       node_address_ack[req][size].sin_port = htons (port_ack[req]);
       memcpy(&node_address_ack[req][size].sin_addr,hp->h_addr_list[0],hp->h_leng
+ th);
     }
    }

    size++;
    if (size >= MAX_PARTITION_SIZE+1) {
       cerr << "Max partition size (" << MAX_PARTITION_SIZE << ") exceeded\n";
       break;
       }

    }

  // Close file
  from.close();


   // compute the address of the laucher
   if((hp = gethostbyname (launch_host)) == 0)
          {
          cerr << "Cannot find address\n";
          exit(1);
          }
   memset (&launch_address,0, sizeof (struct sockaddr_in));
   launch_address.sin_family = AF;
   launch_address.sin_port = htons (launch_port);
   memcpy (&launch_address.sin_addr, hp->h_addr_list[0], hp->h_length);


   // Creation of the sockets

   // Conversion of the port numbers
   for (req=0;req<2;req++) {
      my_port[req]=ntohs(node_address[req][number].sin_port);
      if (ack) port_ack[req]=ntohs(node_address_ack[req][number].sin_port);

      if ((my_socket[req]=create_socket(&my_port[req],SOCK_DGRAM))==-1){
         cerr << "Node" << number << ": Cannot create reply socket\n";
         exit(1);
         }
      if (ack) {

       // sockets used for ACK messages

       if ((socket_ack[req]=create_socket(&port_ack[req],SOCK_DGRAM))==-1){
         cerr << "Node"<<number<<": Cannot create socket\n";
         exit(1);
         }
       }
   }

//   // connect socket to launcher
//   if (connect(socket_dialogue,&launch_address,sizeof (struct sockaddr_in))==-1)
//       {
//       cerr << "Can not connect socket\n";
//       exit(2);
//       }


   // wait until all nodes are ready
   sync_with_nodes();
   if (vb) cerr << "Node"<<number<<" : All nodes ready\n";
}



// -----------------------------------------------
// - disable_interrupts() - Not used by simulation -
// -----------------------------------------------

int node::disable_interrupts() {
interrupt=0;
return 0; }


// -----------------------------------------------
// - enable_interrupts() - Not used by simulation -
// -----------------------------------------------

int node::enable_interrupts() {
interrupt=1;
return 0; }



// -----------------------------------
// - process_ack() - process ACK message -
// -----------------------------------

inline void node::process_ack(ack_message *mes,int req) {

int ack_nb_awaited;

   ack_nb_awaited=message_nb[req][mes->dest];

   // This allows the node to accept an ACK message which number
   // is higher than expected. It means that the other node has
   // already received the data, and that it is pointless to go
   // on sending old messages
```

```
    while (ack_nb_awaited%ack!=0) ack_nb_awaited++;

    if (vb==2)
        fprintf(stderr,"Node%d :received ACK %d from node %d\n",
        number,mes->number,mes->source);
    if (mes->number == ack_nb_awaited) {
        // number ok
        if (req) is_ack[req][mes->dest]=1;
        else is_ack[req][mes->dest]=1;
        }
    if (mes->number < ack_nb_awaited) {
        // Old ACK message
        // The other node has probably received the data twice
        // because it didn't send the ACK fast enough
        if (vb==2) fprintf(stderr,"Node%d : Received old ACK message (#%d instead
+ of %d)\n"
                ,number,mes->number,ack_nb_awaited);
        }
    if (mes->number > ack_nb_awaited) {
        // the ACK number is to high ?!!
        // This should NEVER happen !!!
        fprintf(stderr,"Node%d: Received bad ACK message (#%d instead of %d)\n",
                number,mes->number,ack_nb_awaited);
        }
}


// -------------------------------------
// - poll_ack() - Poll ack interface(s) -
// -------------------------------------

void node::poll_ack(int req){
    int n;
    int flag=1;
    struct sockaddr_in adr_temp;
    int length=sizeof(struct sockaddr_in);
    ack_message mes;

    // sockets in non-blocking mode

    if (setsocknoblock(socket_ack[req])==-1) {
            cerr << "setsocknoblock()\n";
            exit(1);
            }

    // poll loop

    while(flag) {
        flag=0;

        // checks the ack interface
        n=recvfrom(socket_ack[req],&mes,sizeof(ack_message),0,&adr_temp,&length);

        if (n!=-1){
          if (n!=sizeof(ack_message)) fprintf(stderr,
                "WARNING Node%d: received truncated message (%d out of %d bytes)\n",
                number,n,sizeof(ack_message));
          flag=1;
          process_ack(&mes,req);
          }
    }
}


// -------------------------------------
// - send_ack() - send acknowledgment -
// -------------------------------------

inline void node::send_ack(ack_message *mess,int req) {

    // set socket in blocking mode
    setsockblock(socket_ack[req]);

    // send message
    if (sendto(socket_ack[req],mess,sizeof(ack_message),0,
          &node_address_ack[req][mess->source],sizeof(struct sockaddr_in))==-1) {
          cerr << "sendto\n";
          exit(2);
          }

}


// ------------------------------------------------------
// - send_ack() - send acknowledgment, changing msg number -
// ------------------------------------------------------

inline void node::send_ack(ack_message *mess,int num,int req) {


    mess->number=num;

    // set socket in blocking mode
    setsockblock(socket_ack[req]);

    // send message
    if (sendto(socket_ack[req],mess,sizeof(ack_message),0,
          &node_address_ack[req][mess->source],sizeof(struct sockaddr_in))==-1) {
          cerr << "sendto\n";
          exit(2);
          }
}


// ------------------
// - call_handler() -
// ------------------

inline void node::call_handler(active_message *mes) {

    if (mes->nb_arg==4) mes->handler4(mes->data[0],mes->data[1],mes->data[2],mes->d
+ ata[3]);
      else if (mes->nb_arg==3) mes->handler3(mes->data[0],mes->data[1],mes->data[2]
+ );
        else if (mes->nb_arg==2) mes->handler2(mes->data[0],mes->data[1]);
          else if (mes->nb_arg==1) mes->handler1(mes->data[0]);
            else if (mes->nb_arg==0) mes->handler0();
}


// ---------------------
// - call_lg_handler() -
// ---------------------

inline void node::call_lg_handler(lg_active_message *mes) {

  mes->handler(mes->long_data);

}


// ------------------------------------------------
// - process_lg_msg() - Process a long message -
```

```
// -------------------------------------------

void node::process_lg_msg(lg_active_message *mes,int req) {


 if (!ack) {
    call_lg_handler(mes);
    return;
    }
 else {
      ack_message ack_msg;


      // Initialization of the ACK message
      mes->ack(&ack_msg);

      if (mes->number > message_nb_received[req][mes->source]+ack ) {
          // I missed several messages ?!?
          // The message received has a number which is too high .
          // This message shouldn't have been sent in the first place,
          // because it needs an ACK which has not been sent...
          // This should NEVER happen !!!
          fprintf(stderr,"Node%d: WARNING missed messages (#%d instead of %d)!!\n",
                number,mes->number,message_nb_received[req][mes->source]+1);
          return;
          }

      if (mes->number < message_nb_received[req][mes->source]+1 ) {
          // Old message
          // This happens when the ACK message has not been received

          // send acknowledgment
          send_ack(&ack_msg,message_nb_received[req][mes->source],req);

          if (vb==2) fprintf(stderr,"Node%d: Old msg received (%d instead of %d+) se
+ nding ACK %d\n",
              number,mes->number,message_nb_received[req][mes->source]+1,
              message_nb_received[req][mes->source]);
          return;
          }

      if (mes->number == message_nb_received[req][mes->source]+1+
                counter_msg_received[req][mes->source] )  {
          // Good message
          if (vb==2) fprintf(stderr,"Node%d: long msg #%d received from node %d on n
+ etwork %d\n",
              number,mes->number,mes->source,req);

          waiting=0; // reset poll() counter
          if (resending) resending=0; // stop resending messages

          // Increment the counter of messages received
          // If the counter has the good value, send acknowledgment
          if ( ++counter_msg_received[req][mes->source] == ack) {

            // reset counter
            counter_msg_received[req][mes->source] = 0;

            // Increment the number of messages received
            message_nb_received[req][mes->source]+=ack;

            // send acknowledgment
            if(vb==2) fprintf(stderr,"Node%d: sending ACK %d\n",number,mes->number);
            send_ack(&ack_msg,req);
            }

          // call handler
          call_lg_handler(mes);
          return;
          }

 // If none of these cases is correct, it means that a message was lost
 // so, we just wait for the sending node to send the messages again...
 // TODO ? Send a NACK (non-acknowledged) message ?
 // or it means that the message had been already received
      if(vb==2) fprintf(stderr,"Node%d: received bad msg #%d\n",number,mes->number);

   }
 }


 // -------------------------------------
 // - process_msg() - Process a message -
 // -------------------------------------

 void node::process_msg(active_message *mes,int req) {

 if (!ack) {
    call_handler(mes);
    return;
    }
 else {
      ack_message ack_msg;

      // Initialization of the ACK message
      mes->ack(&ack_msg);

      if (mes->number > message_nb_received[req][mes->source]+ack ) {
          // I missed several messages ?!?
          // The message received has a number which is too high .
          // This message shouldn't have been sent in the first place,
          // because it needs an ACK which has not been sent...
          // This should NEVER happen !!!
          fprintf(stderr,"Node%d: WARNING missed messages (#%d instead of %d)!!\n",
                number,mes->number,message_nb_received[req][mes->source]+1);
          return;
          }

      if (mes->number < message_nb_received[req][mes->source]+1 ) {
          // Old message
          // This happens when the ACK message has not been received

          // send acknowledgment
          send_ack(&ack_msg,message_nb_received[req][mes->source],req);

          if (vb==2) fprintf(stderr,"Node%d: Old msg received (%d instead of %d+) se
+ nding ACK %d\n",
              number,mes->number,message_nb_received[req][mes->source]+1,
              message_nb_received[req][mes->source]);
          return;
          }

      if (mes->number == message_nb_received[req][mes->source]+1+
                counter_msg_received[req][mes->source] )  {
          // Good message
          waiting=0; // reset poll() counter
          if (resending) resending=0; // stop resending messages

          if (vb==2) fprintf(stderr,"Node%d: msg %d received from node %d on network
+  %d\n",
              number,mes->number,mes->source,req);

          // Increment the counter of messages received
          // If the counter has the good value, send acknowledgment
```

```
        if ( ++counter_msg_received[req][mes->source] == ack) {

          // reset counter
          counter_msg_received[req][mes->source] = 0;

          // Increment the number of messages received
          message_nb_received[req][mes->source]+=ack;

          // send acknowledgment
          if(vb==2) fprintf(stderr,"Node%d: sending ACK %d\n",number,mes->number);
          send_ack(&ack_msg,req);
          }

        // call handler
        call_handler(mes);
        return;
        }

// If none of these cases is correct, it means that a message was lost
// so, we just wait for the sending node to send the messages again...
// TODO ? Send a NACK (non-acknowledged) message ?
// or it means that the message had been already received
    if(vb==2) fprintf(stderr,"Node%d: received bad msg #%d\n",number,mes->number);

  }
}


// -------------------------------------------------------------------------
// - test_ack() - Test if ACK was received and resend messages if necessary  -
// -------------------------------------------------------------------------

inline void node::test_ack(int dest,int req) {

   int i;

   // Any pending ACK message ?
   // poll for acknowledgment
   //       poll_ack(req);
   // no, because, message_nb[req][dest] hasn't got the good value (1 more)

   // Should I check if messages where acknowledged ?
   if ( (counter_msg_sent[req][dest]!=0) &&
      ((counter_msg_sent[req][dest])%ack == 0) )
      {
      // Any pending ACK message ?
      // poll for acknowledgment
      poll_ack(req);

      // reset counter
      counter_msg_sent[req][dest]=0;

      // message not acknowledged ?
      while (!is_ack[req][dest]) {
         // a bit arbitrary...
         for(i=0;i<1000;i++) {
               poll_ack(req);
               if (is_ack[req][dest]) break;
               }
         //acknowledgment was received
         if (is_ack[req][dest]) break;

         //acknowledgment was not received
         if (vb)    {
           if (message_sent[req][dest][0].type)
              fprintf(stderr,"Node%d: Must resend messages #%d-%d to node %d\n",
                number,message_sent[req][dest][0].lg_msg.number,
                message_sent[req][dest][0].lg_msg.number+ack-1,dest);
           else
              fprintf(stderr,"Node%d: Must resend messages #%d-%d to node %d\n",
                number,message_sent[req][dest][0].a_msg.number,
                message_sent[req][dest][0].a_msg.number+ack-1,dest);
           }

         // previous messages acknowledged
         is_ack[req][dest]=1;
         // Go back in the counter
         message_nb[req][dest]-=ack;

         // Send messages again
         for (i=0; i<ack; i++) {
             if (message_sent[req][dest][i].type) // long message
                   send_msg(&message_sent[req][dest][i].lg_msg,req);
             else // normal message
                   send_msg(&message_sent[req][dest][i].a_msg,req);
             if (is_ack[req][dest]) break;
             }
         // reset counter
         counter_msg_sent[req][dest]=0;
      }
   }

}



// ---------------------------------------------------------------
// - resend_all_messages() - resend all messages not acknowledged  -
// ---------------------------------------------------------------

void node::resend_all_messages(){

   int i,j,k;

   for (j=0;j<2;j++)
      for (i=0;i<size;i++)
        if (i!=number) test_ack(i,j);


    // Send messages again

   resending=1; // allows the send_msg() function to know that it is a resending

   for (j=0;j<2;j++) {
      for (k=0;k<size;k++) {
        if ( (counter_msg_sent[j][k]==0) ||
           ((counter_msg_sent[j][k])%ack == 0) ) continue; // processed by test_ac
+ k() or no msg

        for (i=0; i<counter_msg_sent[j][k]; i++) {
             if (!resending) return;
             if (message_sent[j][k][i].type) // long message
                   send_msg(&message_sent[j][k][i].lg_msg,j);
             else { // normal message
                   if (vb==2) fprintf(stderr,"Node%d: Resending message %d on netw
+ ork %d to node %d\n",number,
                             message_sent[j][k][i].a_msg.number,j,message_sent
+ [j][k][i].a_msg.dest);
                   send_msg(&message_sent[j][k][i].a_msg,j);
                   }
           }
        }
   }
   resending=0;
```

```
}


// ---------------------------------
// - poll() - Poll both interfaces -
// ---------------------------------

void node::poll(int req){

// req = 1 : request interface
// req = 0 : reply interface
// req = 2 : both interfaces

   int n;
   int flag=1;
   struct sockaddr_in adr_temp;
   int length=sizeof(struct sockaddr_in);
   active_message mes;
   lg_active_message lg_mes;

   // if polling for the 2000th time in a row... resend last messages to ALL nodes
   if (ack) {
        if (waiting==2000) {
           if (vb) fprintf(stderr,"Node%d: polling...\n",number);
           resend_all_messages();
           waiting=0;
           }
        waiting++; // counter of poll() calls
   }

   // sockets in non-blocking mode
   if (req)
     if (setsocknoblock(my_socket[1])==-1) {
        cerr << "setsocknoblock()\n";
        exit(1);
        }
   if (req!=1)
     if (setsocknoblock(my_socket[0])==-1) {
        cerr << "setsocknoblock()\n";
        exit(1);
        }

   // poll loop

   while(flag) {
     flag=0;

     if (req) {
       // checks the request interface
       n=recvfrom(my_socket[1],&lg_mes,sizeof(lg_active_message),0,&adr_temp,&lengt
+ h);

       if (n!=-1){
         if (lg_mes.nb_arg==-1) { // long message indeed...
            if (n!=sizeof(lg_active_message)) fprintf(stderr,
                 "WARNING Node%d: received truncated message (%d out of %d bytes)",
                 number,n,sizeof(active_message));
            flag=1;
            // Process message
            process_lg_msg(&lg_mes,1);
            }
         else  { // normal message
            mes=lg_mes;
            if (n!=sizeof(active_message)) fprintf(stderr,
                 "WARNING Node%d: received truncated message (%d out of %d bytes)",
                 number,n,sizeof(active_message));
            flag=1;
            // Process message
            process_msg(&mes,1);
            }
         }
       }

     if (req!=1) {
       // checks the reply interface
       n=recvfrom(my_socket[0],&lg_mes,sizeof(lg_active_message),0,&adr_temp,&lengt
+ h);


       if (n!=-1){
         if (lg_mes.nb_arg==-1) { // long message indeed...
            if (n!=sizeof(lg_active_message)) fprintf(stderr,
                 "WARNING Node%d: received truncated message (%d out of %d bytes)",
                 number,n,sizeof(active_message));
            flag=1;
            // Process message
            process_lg_msg(&lg_mes,0);
            }
         else { // normal message
            mes=lg_mes;
            if (n!=sizeof(active_message)) fprintf(stderr,
                 "WARNING Node%d: received truncated message (%d out of %d bytes)",
                 number,n,sizeof(active_message));
            flag=1;
            // Process message
            process_msg(&mes,0);
            }
         }
       }
     }
}



// ---------------------------------------
// - send_msg() - send an active message -
// ---------------------------------------

void node::send_msg(active_message *mess,int req) {


   if (ack) test_ack(mess->dest,req);
   waiting=0; // reset poll() counter
   // set socket in blocking mode
   setsockblock(my_socket[req]);

   // send message
   if (sendto(my_socket[req],mess,sizeof(active_message),0,
           &node_address[req][mess->dest],sizeof(struct sockaddr_in))==-1) {
      cerr << "sendto\n";
      exit(2);
      }

   if ((ack)&&(!resending)) {
     // increment the number of messages sent
     message_nb[req][mess->dest]++;

     // store the message
     message_sent[req][mess->dest][counter_msg_sent[req][mess->dest]].type=0;
     message_sent[req][mess->dest][counter_msg_sent[req][mess->dest]].a_msg=*mess;

     // status: not acknowledged yet
     is_ack[req][mess->dest]=0;
```

```
     // increment counter
     counter_msg_sent[req][mess->dest]++;
     }

   if (vb==2) fprintf(stderr,"Node%d: Message #%d sent to node%d on network %d\n",
       number,mess->number,mess->dest,req);

   // poll interfaces for incoming messages
   if (req) poll(2); //poll both interfaces
   else poll(0);     //poll only reply interface

}



// -------------------------------------------
// - send_msg() - send a long active message -
// -------------------------------------------

void node::send_msg(lg_active_message *mess,int req) {

   if (ack) test_ack(mess->dest,req);
   waiting=0; // reset poll() counter

   // set socket in blocking mode
   setsockblock(my_socket[req]);
   // send message
   if (sendto(my_socket[req],mess,sizeof(lg_active_message),0,
            &node_address[req][mess->dest],sizeof(struct sockaddr_in))==-1) {
      cerr << "sendto\n";
      exit(2);
      }

   if ((ack)&&(!resending)) {
    // increment the number of messages sent
    message_nb[req][mess->dest]++;

    // store the message
    message_sent[req][mess->dest][counter_msg_sent[req][mess->dest]].type=1;
    message_sent[req][mess->dest][counter_msg_sent[req][mess->dest]].lg_msg=*mess;

    // status: not acknowledged yet
    is_ack[req][mess->dest]=0;

    // increment counter
    counter_msg_sent[req][mess->dest]++;
    }

   if (vb==2) fprintf(stderr,"Node%d: Long message #%d sent to node %d on network
+ %d\n",
       number,mess->number,mess->dest,req);


   // poll interfaces for incoming messages
   if (req) poll(2); //poll both interfaces
   else poll(0);     //poll only reply interface

}

// --------------------------------------------------------------
// - request() - send a request message, long argument handler -
// --------------------------------------------------------------

void node::request(int dest_node,void (*handler)(char *),char *data1) {

   if (sizeof(data1)<=DATA_SIZE) {
      // The message can be sent as is
      lg_active_message mess(number,dest_node,message_nb[1][dest_node]+1,0,handler
+ ,(char *)data1);
      send_msg(&mess,1);
      }
   else {

      // This part hasn't been tested yet (and is not working...)

      // The message is cut in smaller parts
      fprintf(stderr,"Message too long ! : %d bytes\n",sizeof(data1));
      int current_size=sizeof(data1);
      int n=0;
      lg_active_message mess(number,dest_node,message_nb[1][dest_node]+1,1,handler
+ );

      while (current_size>DATA_SIZE) {
         memcpy(mess.long_data,(char *)data1+n*DATA_SIZE,DATA_SIZE);
         send_msg(&mess,1);
         current_size-=DATA_SIZE;
         n++;
         }
      // Send last part
      memcpy(mess.long_data,(char *)data1+n*DATA_SIZE,current_size);
      mess.to_be_contd=0;
      send_msg(&mess,1);

   }
}


// ------------------------------------------------------------
// - request() - send a request message, 4 arguments handler -
// ------------------------------------------------------------

void node::request(int dest_node,void (*handler)(int,int,int,int),int data1,int da
+ ta2,int data3,int data4) {

   active_message mess(number,dest_node,message_nb[1][dest_node]+1,handler,data1,d
+ ata2,data3,data4);

   send_msg(&mess,1);
}

// ------------------------------------------------------------
// - request() - send a request message, 3 arguments handler -
// ------------------------------------------------------------

void node::request(int dest_node,void (*handler)(int,int,int),int data1,int data2,
+ int data3) {

   active_message mess(number,dest_node,message_nb[1][dest_node]+1,handler,data1,d
+ ata2,data3);

   send_msg(&mess,1);
}

// ------------------------------------------------------------
// - request() - send a request message, 2 arguments handler -
// ------------------------------------------------------------

void node::request(int dest_node,void (*handler)(int,int),int data1,int data2) {

   active_message mess(number,dest_node,message_nb[1][dest_node]+1,handler,data1,d
+ ata2);

   send_msg(&mess,1);
```

```
}

// ------------------------------------------------------------
// - request() - send a request message, 1 argument handler -
// ------------------------------------------------------------

void node::request(int dest_node,void (*handler)(int),int data1) {

   active_message mess(number,dest_node,message_nb[1][dest_node]+1,handler,data1);

   send_msg(&mess,1);
}

// ------------------------------------------------------------
// - request() - send a request message, 0 argument handler -
// ------------------------------------------------------------

void node::request(int dest_node,void (*handler)()) {

   active_message mess(number,dest_node,message_nb[1][dest_node]+1,handler);

   send_msg(&mess,1);
}

// ------------------------------------------------------------------------------
// - rpc() - send a Remote Procedure Call (RPC) message, long argument handler -
// ------------------------------------------------------------------------------

void node::rpc(int dest_node,void (*handler)(char *),char *data1) {
// WARNING :
//      for now, rpc=request...
request(dest_node,handler,data1);
}


// ---------------------------------------------------
// - rpc() - send a RPC message, 4 arguments handler -
// ---------------------------------------------------

void node::rpc(int dest_node,void (*handler)(int,int,int,int),int data1,int data2,
+ int data3,int data4) {
// WARNING :
//      for now, rpc=request...
request(dest_node,handler,data1,data2,data3,data4);
}

// ---------------------------------------------------
// - rpc() - send a RPC message, 3 arguments handler -
// ---------------------------------------------------

void node::rpc(int dest_node,void (*handler)(int,int,int),int data1,int data2,int
+ data3) {
// WARNING :
//      for now, rpc=request...
request(dest_node,handler,data1,data2,data3);
}

// ---------------------------------------------------
// - rpc() - send a RPC message, 2 arguments handler -
// ---------------------------------------------------

void node::rpc(int dest_node,void (*handler)(int,int),int data1,int data2) {
// WARNING :
//      for now, rpc=request...
request(dest_node,handler,data1,data2);
}
```

```
// --------------------------------------------------
// - rpc() - send a RPC message, 1 argument handler -
// --------------------------------------------------

void node::rpc(int dest_node,void (*handler)(int),int data1) {
// WARNING :
//      for now, rpc=request...
request(dest_node,handler,data1);
}

// --------------------------------------------------
// - rpc() - send a RPC message, 0 argument handler -
// --------------------------------------------------

void node::rpc(int dest_node,void (*handler)()) {
// WARNING :
//      for now, rpc=request...
request(dest_node,handler);
}



// ----------------------------------------------------------
// - reply() - send a reply message, long argument handler -
// ----------------------------------------------------------

void node::reply(int dest_node,void (*handler)(char *),char *data1) {

   if (sizeof(data1)<=DATA_SIZE) {
      // The message can be sent as is
      lg_active_message mess(number,dest_node,message_nb[0][dest_node]+1,0,handler
+ ,(char *)data1);
      send_msg(&mess,1);
      }
   else cerr << "WARNING: Message too long !!\n";
}



// ------------------------------------------------------
// - reply() - send a reply message, 4 arguments handler -
// ------------------------------------------------------

void node::reply(int dest_node,void (*handler)(int,int,int,int),int data1,int data
+ 2,int data3,int data4) {

   active_message mess(number,dest_node,message_nb[0][dest_node]+1,handler,data1,d
+ ata2,data3,data4);

   send_msg(&mess,0);
}


// ------------------------------------------------------
// - reply() - send a reply message, 3 arguments handler -
// ------------------------------------------------------

void node::reply(int dest_node,void (*handler)(int,int,int),int data1,int data2,in
+ t data3) {

   active_message mess(number,dest_node,message_nb[0][dest_node]+1,handler,data1,d
+ ata2,data3);

   send_msg(&mess,0);
}

// ------------------------------------------------------
```

```
// - reply() - send a reply message, 2 arguments handler -
// -------------------------------------------------------

void node::reply(int dest_node,void (*handler)(int,int),int data1,int data2) {

   active_message mess(number,dest_node,message_nb[0][dest_node]+1,handler,data1,d
+ ata2);

   send_msg(&mess,0);
}

// -----------------------------------------------------
// - reply() - send a reply message, 1 argument handler -
// -----------------------------------------------------

void node::reply(int dest_node,void (*handler)(int),int data1) {

   active_message mess(number,dest_node,message_nb[0][dest_node]+1,handler,data1);

   send_msg(&mess,0);
}

// -----------------------------------------------------
// - reply() - send a reply message, 0 argument handler -
// -----------------------------------------------------

void node::reply(int dest_node,void (*handler)()) {

   active_message mess(number,dest_node,message_nb[0][dest_node]+1,handler);

   send_msg(&mess,0);
}


// ---------------------------------------------------
// - sync_with_nodes() - synchronization of the nodes -
// ---------------------------------------------------

void node::sync_with_nodes() {
   char mes[DIALOGUE_SIZE];
   int n;

   // The dialogue socket should always be in blocking mode
   // -> no need to put it in blocking mode (default)

   // to change if launcher only used for sync_with_node
   // ( DIALOGUE_SIZE -> 1 )
   // init. message to send
   memset(mes,'@',DIALOGUE_SIZE);

   // send message to launcher
   if (sendto(socket_dialogue,mes,DIALOGUE_SIZE,0,
            &launch_address,sizeof(struct sockaddr_in))==-1) {
      cerr << "can not send\n";
      exit(2);
      }

   // wait for the answer
   n=recv(socket_dialogue,mes,1,0);
   if (mes[0]!='@') {
      cerr << "Unknown message received\n";
      exit(1);
      }

   if (vb) cerr << "Node"<<number<<": sync_with_node() done\n";
}
```

```
// ---------------------------------------
// - fset_io_mode() - does nothing for now -
// ---------------------------------------

int node::fset_io_mode(FILE *, int) {
  return 0;
}
```

```
/* go.cc - launch the simulation
 *
 * History:
 *   05/16/95 - Created by N.Guerin
 *   08/08/95 - Last modification by N.Guerin
 */


#include "node.h"

extern "C" {
#include <sys/wait.h>
}

#define HOSTNAME_SIZE   200
#define LAUNCH_PORT     7389  // default port# of the launcher


struct sockaddr_in node_address[MAX_PARTITION_SIZE]; // addresses of all nodes
int launch_socket; // launcher socket
int size=0;         // partition size = number of entries in HOSTS
int vb=0;           // verbose mode off
int ack=0;          // acknoledgement mode off
int tofile=0;       // write to file off
pid_t process[MAX_PARTITION_SIZE]; // pid of the launching processes (not used)
int nb_finished=0;   // number of processes terminated


// ----------------------------------------------
// - main_loop() - Main loop of the program -
// -      Allows display of messages on the -
// -      screen and sync_with_nodes()      -
// ----------------------------------------------

void main_loop() {

   struct sockaddr_in adr_temp;
   int length=sizeof(struct sockaddr_in);
   char mes[DIALOGUE_SIZE];
   int n,i;
   int counter=0;
   char answer='@';

   for(;;) {
      memset(mes,0,DIALOGUE_SIZE);
      // Wait for message, and read it
      n=recvfrom(launch_socket,mes,DIALOGUE_SIZE,0,&adr_temp,&length);
      if (vb==2) cout << "Launcher: Message received from port "<< adr_temp.sin_po
+ rt << " \n";

      // Test if sync_with_node message
      if (mes[0]=='@') { // sync_with_node
        counter++;
         if (counter==size) { // all nodes are waiting
          // reset counter
          counter=0;
          // send signal to all nodes
          for (i=0;i<size;i++) {
                if (sendto(launch_socket,&answer,1,0,&node_address[i],
                                     sizeof(struct sockaddr_in))==-1) {
                   cerr << "sendto\n";
                   exit(2);
                 }
          }
          if (vb) cout << "Launcher: Sync_with_node() succeded\n";
        }
      }
      else if (mes[0]!=0) {  // simple message. In fact, not implemented...
         cout << "message: " << mes << '\n';fflush(stdout);
      }
   }
}


// ------------------------------------------------------------
// - create_socket() - General function that creates a socket -
// ------------------------------------------------------------

static int create_socket(int *port,int type)
{
        int desc; /* descriptor of the new socket */
        struct sockaddr_in name; /* address of the socket */
        int length; /* Length of address */

        /* Creation of the socket */
        if ((desc=socket(AF,type,0))==-1){
                cerr << "Cannot create socket\n";
                exit(2);
                }
        memset((char *)&name,0,sizeof(name));
        name.sin_port=*port;
        name.sin_addr.s_addr=INADDR_ANY;
        name.sin_family=AF;
        if(bind(desc,&name,sizeof(name))){
                fprintf(stderr,"go: Can not bind socket\n");fflush(stderr);
                exit(2);}
        length=sizeof(name);
        if(getsockname(desc,&name,&length)){
                perror("Cannot get socket name\n");
                exit(2);
                }
        *port=ntohs(name.sin_port);
        return(desc);
}

// ------------------------------
// Handler for SIGCHLD signal
// ------------------------------

void node_finished(int)
{
//fprintf(stderr,"Node :SIGNAL %d caught, one node is finished\n",toto);
//fflush(stderr);
(void) signal(SIGCHLD,SIG_IGN);
(void) signal(SIGCHLD, node_finished);
nb_finished++;
if (nb_finished==size) exit(0);
}

// ----------
// - main() -
// ----------

main(int argc,char **argv)
{
  int launch_port=LAUNCH_PORT;
  char hostname[HOSTNAME_SIZE];
  char argu[STRING_SIZE];
  char argu_buf[STRING_SIZE];
  char login[STRING_SIZE];
  char remote_shell[STRING_SIZE];
  char launch_cmd[STRING_SIZE];
  int i;
```

```
(void) signal(SIGCHLD, node_finished);

  memset (argu,0,STRING_SIZE);
  if (argc!=1) {
     int argc_init=1;
     int flag=1;

     while ((flag)&&(argc_init<argc)) {

        flag=0;

        // Want some help ?
        if (!strcmp(argv[argc_init],"-h")) {
           cout << "usage: " << argv[0] << " [-port nnnn] [-v 1/2] [-ack nnn] [-to
+ file] [node_arguments]\n";
           exit(0);
           }

        // Change port ?
        if (!strcmp(argv[argc_init],"-port")) {
           launch_port=atoi(argv[argc_init+1]);
           argc_init+=2;
           flag=1;
           continue;
           }

        // Verbose mode ?
        if (!strcmp(argv[argc_init],"-v")) {
           vb=atoi(argv[argc_init+1]);
           if ((vb!=1)&&(vb!=2)) vb=0;
           argc_init+=2;
           flag=1;
           continue;
           }

        // Acknowledgment mode ?
        if (!strcmp(argv[argc_init],"-ack")) {
           ack=atoi(argv[argc_init+1]);
           if (ack<1) {
              cerr << "size of ack missing \n";
              exit(1); }
           argc_init+=2;
           flag=1;
           continue;
           }

        // Write to file ?
        if (!strcmp(argv[argc_init],"-tofile")) {
           tofile=1;
           argc_init++;
           flag=1;
           continue;
           }

        }

     // Copy argv in string argu
     for (i=argc_init;i<argc;i++) {
        sprintf(argu_buf,"%s %s",argu,argv[i]);
        strcpy(argu,argu_buf);
        }
     }

  if (gethostname(hostname,HOSTNAME_SIZE)==-1) {
        cerr << "Can not get host name\n";
        exit(1);
        }

  // Open socket
  if ((launch_socket=create_socket(&launch_port,SOCK_DGRAM))==-1){
        cerr << "Cannot create dialogue socket\n";
        exit(1);
        }


  // Read the datas from file HOSTS

  // Open file HOSTS
  ifstream from("HOSTS");
  if (!from) {
     cerr << "Cannot open file HOSTS\n";
     exit(1);
     }


  // Read the file
  char comment;
  char *buf = new char[STRING_SIZE];
  char *host_buf = new char[STRING_SIZE];
  char *name_buf = new char[STRING_SIZE];
  char *path_buf = new char[STRING_SIZE];
  struct hostent *hp;
  int  port_req,port_rep,port_dialogue;

  size=0;

  while (from.getline(buf,STRING_SIZE)) {

     // forget it if it is a comment, or if it is empty
     comment=buf[0];
     if ((comment=='#')||(comment==' ')||(comment=='\n')||(comment==0))
          continue;
     // reads the datas
     if (!sscanf(buf,"%s %d %s %s %s %s",host_buf,&port_req,
        remote_shell,login,name_buf,path_buf)) continue;
     port_rep=port_req+2;
     port_dialogue=port_req+4;
     // compute addresses of the other nodes
     if((hp = gethostbyname (host_buf)) == 0)
            {
            cerr << "Cannot find address\n";
            exit(1);
            }
     memset (&node_address[size],0, sizeof (struct sockaddr_in));
     node_address[size].sin_family = AF;
     node_address[size].sin_port = htons (port_dialogue);
     memcpy (&node_address[size].sin_addr, hp->h_addr_list[0], hp->h_length);

     size++;
     if (size >= MAX_PARTITION_SIZE+1) {
        cerr << "Max partition size (" << MAX_PARTITION_SIZE << ") exceeded\n";
        break;
        }

     if (!strcmp(remote_shell,"rsh"))
        sprintf(launch_cmd,"%s -l %s %s",remote_shell,login,host_buf);
     else
        sprintf(launch_cmd,"%s %s -l %s",remote_shell,host_buf,login);

     // Launch program on specified host

     if (tofile)
        sprintf(buf,"%s \'cd %s; ./%s %s %d %s %d %d %d>& result%d\'",launch_cmd,
```

```
                    path_buf,name_buf,argu,size-1,hostname,launch_port,vb,ack,size-1);
        else
           sprintf(buf,"%s \'cd %s; ./%s %s %d %s %d %d %d\'",launch_cmd,
                    path_buf,name_buf,argu,size-1,hostname,launch_port,vb,ack);

        if (vb) {fprintf(stderr,"%s\n",buf);fflush(stderr);}
//        system(buf);

        // fork process and launch program on child
        if ((process[size]=fork())==0) {  // The child process
           execl("/bin/sh", "sh", "-c",buf, 0);
           }
        if (vb) printf("Process %d launched pid=%d\n",size,process[size]);fflush(stdo
+ ut);

        }

     from.close();
     // Main loop...
     if (vb) cout << "Launcher in main loop\n";
     main_loop();
   }
```

```
INCLUDE = -I.
EXTRALIBRARY=   -L.
LIBS=           -lcmmd
#For SUN stations, add -lg++ in LIBS

C++ = CC
#C++ = g++

CFLAGS=   +O2
#CFLAGS=                 -D_SUN -O2
#For SUN stations, add -D_SUN in CFLAGS

################ END OF CONFIGURATION OPTIONS ################


LIBRARIES= $(EXTRALIBRARY) $(LIBS)

lib:    timer.o node.o active_msg.o lg_act_msg.o cmmd.o
        ar ruv libcmmd.a timer.o active_msg.o lg_act_msg.o node.o cmmd.o
        ranlib libcmmd.a > /dev/null
# ranlib useful for SUN stations only

timer.o:    timer.h
        $(C++) -c $(CFLAGS)  timer.cc

active_msg.o:   active_msg.h
        $(C++) -c $(CFLAGS)  active_msg.cc

lg_act_msg.o:   lg_act_msg.h
        $(C++) -c $(CFLAGS)  lg_act_msg.cc

node.o:   timer.h node.h
        $(C++) -c $(CFLAGS)  node.cc

cmmd.o:   timer.h node.h cmmd.h
        $(C++) -c $(CFLAGS)  cmmd.cc

main.o:   timer.h node.h cmmd.h
        $(C++) -c $(INCLUDE) $(CFLAGS)  main.cc

go: node.h go.cc
        $(C++) -o go $(CFLAGS) go.cc $(LIBRARIES)

all: lib go
```

```
/* main.cc
 */

#include <cmmd.h>
#include <iostream.h>

extern "C" {
#include <stdlib.h>
#include <stdio.h>
}

#define MAX 4


int stop=0;
int counter=0;

// --------------------
// - hand() : Handler -
// --------------------

void hand(int data1)
{
int i=CMMD_self_address();

fprintf(stderr,"Node%d: received data %d\n",CMMD_self_address(),data1);fflush(stde
+ rr);

if (data1==MAX) {
        stop=1;
        if (i==0) return;
        }

if (i==0) data1++;

i++;
if (i==CMMD_partition_size()) i=0;
CMAML_request(i,hand,data1);
}



main(int argc,char **argv)  // argc and argv must be present here
{

// to add if you are using the simulation; needs stdlib.h (atoi())
// begin
{
  // test number of arguments
  if (argc<6) {
     fprintf(stderr,"You must launch the CMMD simulation to execute this program\n
+ ");
     exit(2);
     }

  // Init with the 5 last arguments
  CMMD_init_simulation(atoi(*(argv+argc-5)),*(argv+argc-4),atoi(*(argv+argc-3)),
        atoi(*(argv+argc-2)),atoi(*(argv+argc-1)));

  // Ignore the 5 last arguments
  argc-=5;
}
// end

CMMD_node_timer_clear(0);
CMMD_node_timer_start(0);
if (CMMD_self_address()==0)
   {
   CMAML_request(1,hand,counter);
   while (!stop) {
        CMAML_poll();
        }
   }
else
   {
   while (!stop) {
        CMAML_poll();
        }
   }


CMMD_node_timer_stop(0);

fprintf(stderr,"Node%d:  Time elapsed:%d sec\n",CMMD_self_address(),CMMD_node_time
+ r_elapsed(0));fflush(stderr);
exit(0);
}
```

```
EXTRALIBRARY= -L../lib
LIBS=           -lcmmd
# -lg++

C++ = CC
CFLAGS=   +O2 -I../include


################ END OF CONFIGURATION OPTIONS #################


LIBRARIES= $(EXTRALIBRARY) $(LIBS)

main:  main.o
	$(C++) -o main $(CFLAGS) main.o $(LIBRARIES)


main.o:
	$(C++) -c $(CFLAGS)  main.cc
```

```
# Host        port       Command  Login  Name and Location of binary file
  tenedos     12267      remsh    guerin main     /tmp_mnt/auto/RAID1_0f/gaudiot/us
+ ers/guerin/project/CMMDsim-1.0/example
  hyde        12365      remsh    guerin main     /tmp_mnt/auto/RAID1_0f/gaudiot/us
+ ers/guerin/project/CMMDsim-1.0/example
  chryse      10248      remsh    guerin main     /RAID1_0/gaudiot/users/guerin/pro
+ ject/CMMDsim-1.0/example
```