Distributed Implementation of a Linda Kernel *

Rogelio ALVEZ †
e-mail: d80403@buevm1.iinus1.ibm.com

Sergio YOVINE ‡
e-mail: sergio@granier.imag.fr

March 11, 1991

Abstract

A computer network is an interconnected collection of autonomous computers. A distributed system consists of a collection of concurrent, autonomous, descentralized and loosely coupled components. Linda is a high-level parallel programming language intended for distributed programming. It consists of a reduced set of primitives which can be embedded in any sequential language to transform it in a parallel one. This work describes a Linda kernel developed for a 3B Computer Network at ESLAI.

Contents

*This work was developed when the authors did the last year of their studies at Escuela Superior Latinoamericana de Informática (ESLAI)

†Computer Research and Advanced Applications Group. IBM Argentina S.A. (Piso 7) Ing. Butty 275 - 1300 Buenos Aires - Argentina.

‡LGI-IMAG. BP 53X - 38041 Grenoble - France.

List of Tables

1 Introduction

A *computer network* [11] is an *interconnected* collection of *autonomous* computers, usually called nodes. Two machines are said to be interconnected if they are able to exchange information. By requiring a computer to be autonomous we mean that it has complete control of its functioning.

Computer networks make distributed applications feasible. A distributed system consists of a collection of concurrent, autonomous, descentralised and loosely coupled components.

Very often, network operating systems [12] do not provide high-level interprocess communication facilities. In fact, building distributed applications even if feasible, becomes a hard work which requires an expert knowledge of the low-level primitives of the network interface.

Linda [2,3,4,5,6] is a high-level parallel programming language intended for distributed programming. It consists of a reduced set of primitives over a virtual, associative shared memory,

called *tuple space*, which can be embedded in any sequential programming language to transform it in a parallel language.

Linda has some interesting features which make it adequate within a computer network environment. Firstly, in a distributed system services are required by name rather than by location [10]. Secondly, the operations of Linda can be embedded in any language. This allows, at least in principle, to implement the different components of a distributed application in different programming languages, choosing the more appropriate one for each task.

This work describes a Linda kernel developed for a 3B Computer Network[1] at ESLAI. The structure of this paper is as follows. In Section 2, we present the operations of Linda. We discuss the generative communication model and the programming style supported by Linda. In Section 3, we brefly study an operational semantics of Linda. In Section 4, we discuss several alternatives to design a run-time system for Linda in a computer network. In Section 5, we present the implementation developed.

2 Linda

Linda is a reduced set of simple operations intended for distributed and parallel programming [2,3,4,5,6]. Linda is based on a tuple space memory which supports generative communication.

2.1 Tuple Space

A tuple space is an associative memory which storage unit is the tuple. A tuple can be added, removed or read using the out, in or rd operations, respectively. Tuples are addressed via any collection of its values. This mechanism is called *structure naming*.

Linda has the following primitives:

- out(t) is an asynchronous operation that adds tuple t to the tuple space. The executing process continues immediately.
- in(s) is a synchronous operation that removes from the tuple space a tuple t matching template s. If many tuples match, only one of them is removed. If there is not any matching tuple, then the executing process waits for one to be added to the tuple space.
- rd(s) is the same as in(s), except that the tuple matching template s is not removed from the tuple space, but only read.
- eval(t) is the mechanism provided to create processes. It adds an *unevaluated* tuple to the tuple space and creates a process to evaluate it. Tuples created by eval are called active tuples, and play the role of processes in the generative communication model.

[1] 3B is a trade mark of AT&T.

Since the eval operation is quite complex and its semantics is not very clear, we will consider only the first three operations.

2.2 Generative Communication

Many concurrent processes encompassing a tuple space communicate by adding, removing and reading tuples. A process with data to communicate adds a tuple to the tuple space. A process requiring the data removes or reads the tuple. This mechanism is called *generative communication*.

Generative communication supports:

a. *time uncoupling*, in the sense that non coexisting processes can communicate via the tuple space (i.e. the tuple space is independent of the processes and always exist).

b. *space uncoupling*, since any process can read or remove tuples generated by any other process.

In fact, these properties are common to every shared-memory model of communication.

2.3 Programming in Linda

Linda supports a style of programming based on distributed data structures [6,4], that is data structures that can be accessed by many processes in parallel. The control of such a data structure is not centralized in a single process, but distributed among all those that use it. This property is called *distributed sharing*[2].

There are many examples of distributed data structures. For instance, the bag of tasks, which is the basis of the master-worker paradigm of parallel programming [3,4]. A bag can be easily implemented in Linda as a collection of tuples of the form ("bag",*Atask*). Any process can insert a task in the bag executing out("bag",*thistask*), or remove one from it via in("bag",var *newtask*).

Other interesting example is the distributed array. In Linda, an array is a collection of tuples of the form (*index*,*value*), and an additional tuple ("length",*integer-value*) to denote the length of the array. Assuming that the values range over the integers, an implementation of this data structure is shown in table 1.

This Linda implementation of a distributed array is simpler than the one presented in [13,14]. Also notice that the implementation of variable length arrays is straightforward.

```
proc create(n: integer);
(* to create an array of length n *)
(* 'any' is a special value matching all types *)
     var i: integer;
     begin
          for i:=1 to n do out(i,any);
          out("length",n)
     end

proc put(i,v: integer);
(* to assign value v to element i, provided 1≤i≤n *)
     var j: integer;
     begin
          in(i,var j);
          out(i,v)
     end

proc get(i: integer): integer;
(* to read the value of element i *)
     var v: integer;
     begin
          rd(i,var v);
          return v
     end

proc length(): integer;
(* to know the length of the array *)
     var l:integer;
     begin
          rd("length",var l);
          return l
     end
```

Table 1: Distributed Array

3 Operational Semantics of Generative Systems

A *generative system* consits of a tuple space and a set of concurrent processes performing in, out and rd operations.

The purpose of this section is to describe from an *observational* point of view, the behavior of a generative system.

3.1 Tuple Space

Tuples are the basic elements of information within the shared-memory model of generative communication. In the previous section we have defined informally how a template and a tuple match in the tuple space. We will give a formal description of this operation now.

Let $\mathcal{S} = \{s_1, ..., s_n\}$ be a set of sorts, $\mathcal{C}_{s_i}$ the set of constants of sort s_i, $\mathcal{C} = \bigcup_{i=1}^{n} \mathcal{C}_{s_i}$, and $\mathcal{X}$ a set of variables.

Definition 3.1.1 *Tuple.*
A tuple is defined as $t = (t_1, ..., t_m)$ *where:*
$(\forall i \in \{1, ..., m\})$
i. $t_i = x : s, x \in X \wedge s \in \mathcal{S}$, *or*
ii. $t_i = c : s, c \in \mathcal{C}_s$.

Definition 3.1.2 *Matching.*
Let $t_1 = (x_1 : s_1, ..., x_m : s_m)$ *and* $t_2 = (y_1 : r_1, ..., y_m : r_m)$ *be two tuples. Then, a predicate match(t_1,t_2) is defined as follows:*
$match((x_1 : s_1, ..., x_m : s_m), (y_1 : r_1, ..., y_m : r_m))$
$\quad \equiv_{def} (m = k) \wedge (\forall i \in \{1, ..., m\} : match\_comp(x_i : s_i, y_i : r_i))$
$match\_comp(x : s, y : r)$
$\quad \equiv_{def} (s = r) \wedge ((x \in \mathcal{C}_s \wedge y \in \mathcal{C}_r \wedge x = y) \vee (x \in \mathcal{C}_s \wedge y \in \mathcal{X}) \vee (x \in \mathcal{X} \wedge y \in \mathcal{C}_r))$

Definition 3.1.3 *Tuple Space.*
A Tuple Space is a multiset of tuples. We will use $\oplus$ *to denote the union of multisets.*

3.2 Operational Semantics

In order to modelize generative systems, we will define a simple language whose terms denote sets of parallel processes performing the operations of Linda. Then, we will define an operational semantics for this language by means of a transition relation on terms [1].

3.2.1 Syntax

The syntax is as follows:
$A := in(x) \mid out(x) \mid rd(x)$
$E := nil \mid A.E$
$P := E \mid P||P$

3.2.2 Semantics

Let $q, q', p, p' \in P$ and $\varepsilon, \varepsilon'$ be tuple spaces. We define a transition relation $\rightarrow$ as follows:

(1) $$(out(x).p, \varepsilon) \xrightarrow{out} (p, \varepsilon \oplus \{x\})$$

(2) $$\frac{match(x,y)}{(in(y).p, \varepsilon \oplus \{x\}) \xrightarrow{in} (p, \varepsilon)}$$

(3) $$\frac{match(x,y)}{(rd(y).p, \varepsilon \oplus \{x\}) \xrightarrow{rd} (p, \varepsilon \oplus \{x\})}$$

(4) $$\frac{(p, \varepsilon) \xrightarrow{\alpha} (p', \varepsilon')}{(p||q, \varepsilon) \xrightarrow{\alpha} (p'||q, \varepsilon')}$$

(5) $$\frac{(q, \varepsilon) \xrightarrow{\alpha} (q', \varepsilon')}{(p||q, \varepsilon) \xrightarrow{\alpha} (p||q', \varepsilon')}$$

Finally, we introduce *fairness* as an axiom.

4 Design alternatives

There are two important questions that naturally arise when we want to design a Linda virtual machine [2]:

- How to find the tuples?
- How to store them?

The first is closely related with the criterion adopted to distribute tuple space; the second is related with the choice of required data structures to maintain it.

4.1 Distributing the Tuple Space

Since tuple space is logically similar to a shared memory, the task of building and maintaining it over disjoint-memory hardware - like a local network - represents an interesting implementation problem.

We should place tuple space completely in one node and direct all generative operations to it, but this policy might generate a dangerous "bottleneck", degrading the performance. Furthermore, this solution is not conceptually the most adequate if we want to have an authentic distributed environment. It seems quite clear that we should distribute tuple space in some way over some subset of all nodes. There are several forms to distribute the tuple space. In general, they consist of partitioning the space into small subsets, assigning the managment of each one to a group of nodes. We will brefly discuss here only two extreme cases, in which single nodes are considered instead of groups.

4.1.1 1-N distribution

One possible choice is the following: when a process executes an out(t) operation, the Linda kernel sends a copy of tuple t to every node in the network. Thus, when a process wants a tuple, it only needs to search in its node, because tuple space is completelly replicated in each node. But the attempt to delete it (by means of an in operation) implies the activation of a global protocol, because the tuple must be deleted in all nodes [8]. This option is viable with good performance only if we have great memory capacity and fast communication medium, mainly with reliable broadcast.

4.1.2 N-1 distribution

The inverse choice is also possible. The tuple space can be partitioned into N small parts, each one assigned to one of the nodes of the network. There are several criteria to partition the tuple space:

- *by locality*, tuples are located where they are produced. The out operation is local, while in and rd may cause searching all over the network if the tuple is not locally available. Deleting tuples becomes easier [3]. However, this mechanism could produce overloading of certain nodes, specially those where out operations are very often performed, if not controlled.

- *by a distribution function*, tuples are located according to the value produced by applicating it a hash function. This mechanism can control overloading [9].

In this case, tuples are not replicated. So, its goal is to make an efficient managment of space.

5 Implementation

In this section we will present the Linda kernel implemented for the 3B computer network at ESLAI, consisting of two nodes interconnected via Fthernet.

5.1 Partitioning the Tuple Space

The available hardware did not have enough memory to adopt the 1-N policy discussed in Section 4. Thus, we chose the N-1 alternative with a distribution function criterion. We implemented the tuple space as a great hash table, distributed throughout the network. This table was partitioned in subtables, actually implemented as hash tables, one in each node. On executing an out(t) operation, the Linda kernel will send tuple t to the unique node determined by the application of the hash function to t. Something similar occurs when a tuple s is desired (by means of an in or rd operation). We must ask for that tuple to the node determined by the result of the application of the hash function to s.

Notice that this last policy makes an efficient use of memory, because information is not replicated. Besides, it also minimizes communication, because it is neccesary only one step of communication for an out operation and two steps for in and rd operations (ask and reply).

5.2 Tuples and processes management

The subspace installed in each node is managed by an "endless" process, independent of the application. This process was called "Manager" (table 2).

Manager is encouraged to assign a particular tuple (generated by an out) to a template (generated by an in or rd). The applications' requirements are coded in messages and transmited through the network. Each message includes the operation type, the tuple and the identifier of the requesting process. The message is sended to the node determined by the "space-partition" function (see table 3).

When a process executes an in or rd operation, it sends the request by a message and blocks waiting for the reply. Each in or rd may or not find a matching tuple. In the first case, the manager replies inmediatelly with the found tuple. In the latter, the manager stores the message until a matching tuple arrives. When a new tuple arrives (via out), the manager searchs for delayed matching templates waiting for that tuple. If no messages containing matching templates are found, the tuple is stored in the local space; otherwise the messages are replied. This strategy prevents starvation.

```
proc Manager();
    var te: Tuple_space;
    var se, s: set of Mge;
    var t1, t2: Tuple;
    var op1, op2: Opr;
    var pid1, pid2: ProcId;
    loop forever
        receive((pid1,op1,t1));
        case op1 do
        in:  if ∃t ∈ te such that match(t,t1)
             then te:=te ⊖ t; send((Manager,_,t),pid1);
             else es:=es ⊕ (pid1,op1,t1);
        rd:  if ∃t ∈ te such that match(t,t1)
             then send((Manager,_,t),pid1);
             else es:=es ⊕ (pid1,op1,t1);
        out:
             s:=set of mges in es waiting for t1;
             es:=es ⊖ s;
             for (pid2,rd,t2) ∈ s do send((Manager,_,t1),pid2);
             if ∃ (pid2,in,t2) ∈ s
             (* there is at most one *)
             then send((Manager,_,t1),pid2);
             else et:=et ⊕ t1;
        end case
    end loop
end proc
```

Table 2: The "Manager" process

```
type Opr = enum  in, rd, out ;
type Mge = record pi: ProcId; op: Opr, t: Tuple end;

in(t):
     send((process_id,in,t),dest(t));
     receive((Manager,_,t'));
     instantiate(t,t')

rd(t):
     send((process_id,rd,t),dest(t));
     receive((Manager,_,t'));
     instantiate(t,t')

out(t):
     send((process_id,out,t),dest(t))
```

Table 3: in, rd and out operations

5.3 User's interface

A library implementing the operations of Linda is provided. It can be included in any C program. The operations have a list of parameters. The first element of the list is a string defining the type of the tuple. The rest of the list contains the tuple. Each element of a tuple can be of type string (indicated by an "s"), integer ("i") or double ("f"). Formal parameters (i.e. variables), must be preceded by a symbol "&". For instance, in("sii&i","pepe",1,1,&val), corresponds to an invocation of the in operation with a template of length four, which first argument is the string "pepe", the second and the third are the integer 1, and the last component is a formal argument of type integer. The variable "val", must be defined in the program performing the operation.

5.4 Internal representation of tuples

A tuple is implemented as a structure containing the length of the tuple and an array of components. In fact, the length of the tuples has a maximum. This decision was adopted in order to be able to transmit tuples in a single Ethernet packet (up to 1500 bytes [11]).

Each component has three fields: its type, if it is a variable or a value, and its value (which is undefined if it is a variable).

6 Conclusions

The design of a run time support must include not only the tuples but also the processes that are communicating via tuple space. Since we did not consider active tuples, we decided to exploit the multiprogramming environment provided by UNIX. Thus, we designed the run time support in order to let Linda processes be in two coupling levels simultaneously: multiple processes running in one node and multiple nodes (in practice just two) communicating via Ethernet. Linda permits an uncoupled programming style. Therefore, it is not necessary to determine a priori the number of processes that participate in any application. Whatever is the number of them, they can be created at any time and in any of both nodes; there exist just hardware limitations.

"Manager" processes are installed at the system "start-up" time. The entire system has been implemented in C. For network management we have used the network interface provided by the operating system.

The kernel is totally independent of the number of nodes in the network. Nowadays, it is working with two nodes, but it has been designed to work in the same way with one node or more. It is only necessary to actualize a special system table when the hardware configuration changes.

In the implemented scheme, all tuple requirements involve the use of the communication channel, even though the process that requires a tuple and the manager of the subtable where it is placed are in the same node. These kind of requests could be avoided if tuple subspaces in each node were shared by their managers and local processes. The main advantage is that in this new scheme each manager is only responsible for the administration of external requests. However, the manager and the local processes must compite for the use of the subspace. We believe that this scheme will substantially improve the performance of the applications, because of the reduction in the use of the communication bus. This improvement can be achieved using the IPC facilities (shared memory, semaphores and messages) of UNIX.

7 Acknowledgements

We would like to thank Nick Carriero, who sent us a lot of papers about Linda, and for his suggestions and advise. Daniel Yankelevich made important criticisms and suggestions. Jorge Aguirre gave us his assitance and direction.

References

[1] R. Milner. "Communication and Concurrency". Prentice Hall Inc., C.A.R. Hoare, Series Editor, 1989.

[2] D. Gelernter. "Generative Communication in Linda". ACM Transactions on Programming Languages and Systems, pages 80-112, January 1985.

[3] S. Ahuja, N. Carriero, D. Gelernter. "Linda and Friends". IEEE Computer, Aug 86, 26-34.

[4] N. Carriero, D. Gelernter. "How to write parallel programs: A guide to the perplexed". Yale University, Dept. of Computer Science, Nov. 88 (to appear, ACM Computer Surveys).

[5] N. Carriero, D. Gelernter. "Linda in context". (to appear, Communications ACM).

[6] D. Gelernter, N. Carriero, S. Chandran, S. Chang. "Parallel programming in Linda". *Proceedings of the International Conference on Parallel Processing*, Aug. 85, 255-263.

[7] Nicholas J. Carriero. "Implementing Tuple Space Machines". PhD thesis, Yale University, New Haven, Connecticut, 1987. Department of Computer Science.

[8] N. Carriero, D. Gelernter. "The S/Net's Linda Kernel". *Proceedings Symp. Operating Systems Principles*, Dec. 85.

[9] R. Bjornson, N. Carriero, D. Gelernter. "The implementation an Performance of Hypercube Linda". Research report YALEU/DCS/RR-690, Mar. 89.

[10] Andrew S. Tanenbaum. "Computer Networks". Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1981.

[11] Andrew S. Tanenbaum. "Computer Networks". Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1989.

[12] A. Tanenbaum, R. Van Renesse. "Distributed Operating Systems". Computing Surveys, 17(4), Dec. 85, 419-470.

[13] W. Horwat, A. Chien, W. Dally. "Experience with CST: Programming and Implementation".

[14] W. Dally, A. Chien. "Object-Oriented Concurrent Programming in CST". SIGPLAN NOTICES 24(4), Apr. 89.

A The C code

We include here the main parts of the implementation.

```
/* red.h */
#include <sys/ni.h>
#include <fcntl.h>

#define NID6CPY(a,b)  (a[0]=b[0],\
                       a[1]=b[1],\
                       a[2]=b[2],\
                       a[3]=b[3],\
                       a[4]=b[4],\
                       a[5]=b[5])

#define NID2CPY(a,b)  (a[0]=b[0],\
                       a[1]=b[1])

#define PK_SIZE 250          /* Packet Length*/

#define NINETPORT "/dev/ni"  /* Network Interface driver of 3B */

typedef char id_port[6];     /* port id */

typedef struct { EI_PORT head;
                 char    data[MAXETPACKET];
               } PACK ;

static id_port ELENA_PORT = {0x69,0x69,0x69,0x69,0x69,0x69} ;

static id_port ESLAI_PORT = {0x71,0x71,0x71,0x71,0x71,0x71} ;

static char APPROT[2] = {0x88,0x88} ;

static short approt = 0x8888;

static int RdPORT;     /* Read Port */

static int WrPORT;     /* Write Port */

static int ErrPORT;    /* Error code for network operations */

static id_port MyPORT;
```

```
/*-------------------------------------------------*/
/*                    MANAGER                      */
#include <stdio.h>
#include "red.h"
#include "mensajes.h"
#include "hash.h"

#define TRUE 1
#define FALSE 0

extern int errno;

main()
{
   mensaje *m,*ReadPORT();

   if (InicET()==-1) {
      printf("ERROR %d InicET() \n",errno);
      exit(1);
   }

   InicKernelRdPORT();
   if (ErrPORT) {
      printf("ERROR %d InicKernelRdPORT() \n",ErrPORT);
      exit(2);
   }

   InicKernelWrPORT();
   if (ErrPORT) {
      printf("ERROR %d InicKernelWrPORT() \n",ErrPORT);
      exit(3);
   }

   while (TRUE) {
      if (!(m=ReadPORT())) {
         printf("ERROR %d ReadPORT() \n",ErrPORT);
         exit(4);
      }
      switch(m->op) {
         case IN : Gin(m);
                   break;
         case OUT: Gout(m);
                   break;
         case RD : Grd(m);
```

```
                    break;
            }
        }
    }

/* rd() operation */
Grd(p_mje)
    mensaje *p_mje;
{
    int h_entry;
    LDT *LDTcte;
    int escape=FALSE;
    mensaje *m,*ConstMJE();

    h_entry = hash(p_mje);
    LDTcte=ET[h_entry].lt;
    while ( !escape && LDTcte->sig ) {
        LDTcte = LDTcte->sig;
        if (match(*(p_mje->t),*(LDTcte->t))) {
            m=ConstMJE(MyPORT,NIL,LDTcte->t);
            WritePORT(m,p_mje->origen);
            free(m);
            escape = TRUE;
        }
    }
    if (escape)
        BorrarMJE(p_mje);
    else
        ET[h_entry].lm.ult=AgregarLDM(ET[h_entry].lm.ult,p_mje);
}

/* in() operation */
Gin(p_mje)
    mensaje *p_mje;
{ ... }

/* out() operation */
Gout(p_mje)
    mensaje *p_mje;
{ ... }

/* ------------------------------------------------------*/
/*                    Network Management                 */
```

```
mensaje *ReadPORT()
{
   PACK Pac_In;
   mensaje *PackToMen();

   ErrPORT = 0;
   if (read(RdPORT,&Pac_In,PK_SIZE+2)==-1) {
      ErrPORT = errno;
      return NULL;
   }
   return PackToMen(Pac_In);
}

WritePORT(m,nodo)
   mensaje *m;
   char *nodo;
{
   PACK *MenToPack();

   return write(WrPORT,MenToPack(m,nodo),PK_SIZE);
}

InicKernelRdPORT()
{
   NI_PORT vpst;
   int fd;
   void KernelPORT();

   ErrPORT = 0;
   if ((fd=open(NINETPORT,O_RDONLY))==-1)
      ErrPORT=errno;
   if (ioctl(fd,NIGETA,&vpst)==-1)
      ErrPORT=errno;
   KernelPORT(vpst.srcaddr);
   vpst.rspq_sz = 8;
   vpst.rcvb_sz = PK_SIZE+2;
   vpst.rcvq_sz = 15;
   vpst.protocol = approt;
   vpst.type = ETHERTYPE;
   if (ioctl(fd,NISETA,&vpst)==-1)
      ErrPORT=errno;
   RdPORT = fd;
}
```

```
InicKernelWrPORT()
{ ... }

void ProcessPORT(str)
char *str;
{
   register int i,j;
   int  id;
   id_port pid;

   sprintf(pid,"%d",getpid());
   str[0] = 0x69;
   str[1] = str[4];
   str[2] = str[5];
   for(i=strlen(pid);i<6;i++)
      pid[i]='1';
   j = 3;
   for(i=0;i<3;i++) {
      sscanf(&pid[2*i],"%2x",&id);
      str[j++] = id;
   }
}

void KernelPORT(str)
char *str;
{
   if (str[5] == 0x9d )
      NID6CPY(str,ELENA_PORT);
   else
      NID6CPY(str,ESLAI_PORT);
}


/*-------------------------------------------------------------*/
/*                              IN                             */
in(va_alist)
va_dcl
{
    mensaje *m,*ConstMJE(),*ReadPort();
    va_list ap;
    char *GetVsort(),*MyPort(),
         *signatura,
         *str;
    dir d[TUPLAMAX];
```

```
 vsorts vsort_cte;
 int i=0;
 int lg;
 tupla *tup;

 tup=(tupla *)malloc(sizeof(tupla));
 va_start(ap);
 signatura = va_arg(ap,char *);
 while (*signatura==' ') signatura++;
 while (*signatura != '\0') {
       signatura = GetVsort(signatura,&vsort_cte);
       switch(vsort_cte) {
             case VALINT: { ... }
             case VALDBL: { ... }
             case VALSTR: { ... }
             case VARINT: { ... }
             case VARDBL: { ... }
             case VARSTR: { ... }
             case V_ERR  : return -1;
       }
       i++;
 }
 tup->lg=i;

m=ConstMJE(MyPort(),IN,tup);
if (WritePort(m,DestPort(tup->comp[0].campo.s))==-1) {
   BorrarMJE(m);
   return -1;
}
else
   BorrarMJE(m);
if (!(m=ReadPort())) {
   BorrarMJE(m);
   return -1;
}
/* Instantiation of formal parameters */
for (i=0;i<m->t->lg;i++)
    switch(m->t->comp[i].sort) {
          case INT : if (d[i].fi != NULL)
                        *(d[i].fi)=m->t->comp[i].campo.i;
                     break;
          case DBL : if (d[i].ff != NULL)
                        *(d[i].ff)=m->t->comp[i].campo.f;
                     break;
```

```
                case STR : if (d[i].fs != NULL)
                              strcpy(d[i].fs,m->t->comp[i].campo.s);
                           break;
           }
      BorrarMJE(m);
      return 0;
}
```