View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by NASA Technical Reports Server

34034
P. 9

USING PVM TO HOST CLIPS IN DISTRIBUTED ENVIRONMENTS

Leonard Myers
Computer Science Department
California Polytechnic State University
San Luis Obispo, CA 93402
lmyers@calpoly.edu

Kym Pohl
CAD Research Center
California Polytechnic State University
San Luis Obispo, CA 93402
kpohl@calpoly.edu

Abstract

It is relatively easy to enhance CLIPS to support multiple expert systems running in a distributed environment with heterogeneous machines. The task is minimized by using the PVM (Parallel Virtual Machine) code from Oak Ridge Labs to provide the distributed utility. PVM is a library of C and FORTRAN subprograms that supports distributive computing on many different UNIX platforms. A PVM deamon is easily installed on each CPU that enters the virtual machine environment. Any user with *rsh* or *rexec* access to a machine can use the one PVM deamon to obtain a generous set of distributed facilities. The ready availability of both CLIPS and PVM makes the combination of software particularly attractive for budget conscious experimentation of heterogeneous distributive computing with multiple CLIPS executables. This paper presents a design that is sufficient to provide essential message passing functions in CLIPS and enable the full range of PVM facilities.

Introduction

Distributed computing systems can provide advantages over single CPU systems in several distinct ways. They may be implemented to provide one or any combination of the following: parallelism, fault tolerance, heterogeneity, and cost effectiveness. Our interest in distributive systems is certainly for the cost effectiveness of the parallelism they provide. The speedup and economy possible with multiple inexpensive CPUs executing simultaneously make possible the applications in which we are interested, without a supercomputer host. The economy of distributed computing may be necessary for the realization of our applications, but we are even more interested in distributed systems to provide a simplicity of process scheduling and rapid interaction of low granularity domain specific actions. Our approach to implementing certain large and complex computer systems is focused on the notion of quickly providing reasoned response to user actions.

We accomplish the response by simultaneously examining the implications of an user action from many points of view. Most often these points of view are generated by individual CLIPS[1] expert systems. However, there is not necessarily a consistency in these viewpoints. Also, a domain inference may change over time, even a small period of time, as more information becomes available. Our systems have the experts immediately receive and respond to low level user action representations. The action representations encode the lowest level activities that are meaningful to the user, such as the drawing of a line in a CAD environment. Thus, if there are a dozen, or more, expert domains involved in an application, it is necessary to transmit user actions to each expert and combine the responses into a collective result that can be presented to the user. Of course, the intention is to emulate, in a relatively gross manner, the asynchronous parallel cooperative manner in which a biological brain works.

Over the course of eight years, the CAL POLY CAD Research Center and CDM Technologies, Incorporated have developed several large applications and many small experiments based essentially on the same ideas.[2, 3, 4] In the beginning we wrote our own socket code to communicate between the processes. Unfortunately, with each new brand of CPU introduced into

Volume: 2
PRECEDING PAGE BLANK NOT FILMED
PAGE 202 INTENTIONALLY BLANK

a system it was necessary to modify the communication code. There simply are not sufficient standards in this area to provide portability of code, except by writing functions that are machine specific. Moreover, our main interest is not the support of what is essentially maintenance of the inter-process communication code. But for many years it seemed necessary to support our own communication code because the other distributed systems we investigated were simply too inefficient for our applications, even though they were generally much more robust. We then found that PVM[5] code was very compatible with our requirements.

PVM

PVM is a free package of programs and functions that support configuration, initiation, monitoring, communication, and termination of distributed UNIX processes among heterogeneous CPUs. Version 3.3 makes UDP sockets available for communication between processes on the same CPU, which is about twice as fast as the TCP sockets generally necessary for communication between CPUs. Also, shared memory communication is possible on SPARC, SGI, DEC, and IBM workstations, and several multiprocessor machines.

Any user can install the basic PVM daemon on a CPU and make it available to all other users. It does not require any root privileges. Users can execute their own console program, *pvm3*, that provides basic commands to dynamically configure the set CPUs that will make up the 'virtual machine'. The basic daemon, *pvmd*, will establish unique dynamic socket communications for each user's virtual machine. Portability is obtained through the use of multiple functions that are written specifically for each CPU that is currently supported. An environment variable is used to identify the CPU type of each daemon and hardware specific directory names, such as *SUN4*, are used to provide automatic recognition of the proper version of a PVM function. Since all of the common UNIX platforms are currently supported, the code is very portable.

The fundamental requirements for a PVM distributed system involving CLIPS processes consist of starting a CLIPS executable on each selected CPU and calling PVM communication facilities from CLIPS rulesets. The first of these tasks is very easy. PVM supports a 'spawn' command from the console program as well as a 'pvm_spawn' function that can be called from a user application program. Both allow the user to let PVM select the CPU on which each program will be executed or permit the user the specify a particular CPU or CPU architecture to host a program. Over sixty basic functions are supported for application programming in C, C^{++}, or FORTRAN, but only the most basic will be mentioned here and only in their C interface forms.

PVM Communication Basics

Each PVM task is identified by its 'task id'(tid), an integer assigned by the local PVM daemon. The tid is uniquely generated within the virtual machine when either the process is spawned, or the pvm_mytid function is first called. Communication of messages between two processes in a PVM system generally consists of four actions:

1. clear the message buffer and establish a new one	-	**pvm_initsend**
2. pack typed information into the message buffer	-	**pvm_pkint**, **pvm_pkfloat**, etc.
3. send the message to a PVM task or group of tasks	-	**pvm_send**, **pvm_mcast**, etc.
4. receive the message, either blocking or not	-	**pvm_recv**, **pvm_nrecv**, etc.

Each message is required to be assigned an integer, 'msgtype', which is intended to identify its format. For instance, task1 may send a message that consists of ten integers to task2. In the

'pvm_send' used to transmit the message, the last argument is used to identify the type of message. The programmer may decide that such a message is to be of type 4, for example. Then, task2 can execute 'pvm_recv', specifying which task and message type it would like to receive. A -1 value can be used as a wildcard for either parameter in order to accept messages from any task and/or of any type. If task2 indicates that it wishes to accept only a message of type 4 to satisfy this call, it then knows when the 'pvm_recv' blocking function succeeds that it has a message of ten integers in a new input buffer.

As a more complete example, the following code shows a host task that will spawn a client task; then it will send the client a message consisting of an integer and a floating point value; and then it will wait for a reply from the client. The client waits for the message from the host, generates a string message in reply, and halts. When the client message is received, the host prints the message and halts.

Host

```
#include "pvm3.h"
main () {

int  hostid, tids[1];/* pvm encodes pid within a taskid (tid) */
int  ivalue1;       /* first value to be sent               */
float  fvalue2;     /* second value to be sent              */
char message[25];/* string to be returned from client       */

hostid = pvm_mytid;            /* enroll in pvm */

pvm_spawn("Client",     /* name of process to start up  */
          (char**)0,    /* args to be passed to process */
          0,            /* options - 0 to use any CPU   */
          "",           /* host name when not option 0*/
          1,            /* number of copies to spawn    */
          tids);        /* tids of processes started    */

pvm_initsend(PvmDataRaw);  /* get buffer                         */
                           /* no encoding of data if same type CPU */
pvm_pkint(23, 1, 1);       /* pack 1 int into the current buffer  */
pvm_pkfloat(45.678, 1, 1); /* pack 1 float into the current buffer */
pvm_send(tid[0], 2);       /* send buffer as user chosen type 2   */

pvm_recv(-1, 1);           /* wait for message from anyone (-1 is a */
                           /* wildcard) that is user typed as 1     */
pvm_upkstr(message);       /* unpack string from buffer into message*/

printf("I got the message: %s\n", message);

pvm_exit();
}
```

Client

```
#include "pvm3.h"
main() {

int  hostid, clientid;
clientid = pvm_mytid;

pvm_recv(-1, 2);       /* wait for type 2  */
/* we ignore the contents in this example */

pvm_initsend(PvmDataRaw);
pvm_pkstr("Hi Host!"); /* pack string     */
hostid = pvm_parent();    /* get host tid  */
pvm_send(hostid, 1);      /* send string   */
                          /* as type 1     */

pvm_exit();
}
```

Figure 1. A PVM Example

CLIPS Implementation Considerations

The basic need is to assert a fact or template from a rule in one CLIPS process into the factlist of another CLIPS process, which may be executing on a different processor. We might want a function that could use a PVM buffer to transmit a CLIPS 'person' template as in figure 2:

```
( BMPutTmplt ?BufRef ( person  (name  "Len")  ( hair "brown") (type "grumpy")) )
```

Figure 2. Sample Send Template Call

The first problem this presents is that the form of this function call is illegal. It is possible to use a syntax in which the fact that is to be communicated is presented as a sequence of arguments, rather than a parenthesized list. But this syntax does not preserve the appearance of the local assert, which is pleasing to do. The solution is to add some CLIPS source code to provide a parser for this syntax.

Second, consideration must be given as to when the PVM code that receives the messages should execute. It is desirable that receiving functions can be called directly from the RHS of a CLIPS rule. It is also desirable in most of our applications that message templates are transparently asserted and deleted from the receiver's factlist without any CLIPS rules having to fire. In order to accommodate the latter, our CLIPS shell checks for messages after the execution of every CLIPS rule, and blocks for messages in the case that the CLIPS agenda becomes empty.

Third, it is useful to be able to queue message facts to be asserted and have them locally inserted into the receiver's factlist during the same CLIPS execution cycle. There are occasions when a number of rules might fire on a subset of the message facts that are sent. In most cases the best decision as to what rule should fire can be made if all associated message facts are received during one cycle, and all such rules are activated at the same time. Salience can usually be used to select the best of such rules and the execution of the best rule can then deactivate the other alternatives. In order to implement this third consideration, the message facts are not sent for external assertion until a special command is given.

OVERVIEW OF CMS

The CLIPS/PVM interface or CLIPS Message System (CMS), provides efficient, cost effective communication of simple and templated facts among a dynamic set of potentially distributed and heterogeneous clients. These clients may be C, C++, or CLIPS processes. Clients may communicate either single fact objects or collections of facts as a single message. This communication takes place without requiring either the sender or receiver to be aware of the physical location or implementation language of the other. CMS will transmit explicit generic simple fact forms by dynamically building C representations of any combination of basic CLIPS atoms, such as INTEGERs, FLOATs, STRINGs, and MULTIFIELDs. CMS will also communicate facts or templates referenced by a fact address variable.

In addition to the dynamic generic process described above, the communication of templated facts is supported in a more static and execution time efficient manner. Specific routines capable of manipulating client templates in a direct fashion are written for each template that is unique in the number and type of its fields The deftemplate identifies what message form these routines will expect. This is a distinct luxury that eliminates the time involved in dynamically determining the number and types of fields for a generic fact that is to be communicated. The disadvantage is that the routines must be predefined to match the templates to be communicated. However, the applications in which the authors are involved have predetermined vocabularies for the data objects that are to be communicated. In much the same way that deftemplates provide efficiency through the use of predetermined objects, these template-form specific communication routines provide object communication in the most efficient, cost-effective fashion. The following section describes the overall architecture of a system which supports the previously described functionality.

Architecture

The underlying support environment consists of a set of utility managers called the System Manager, Memory Manager, and Error Manager. All three managers provide various

functionality utilized by the communication system to perform efficient memory usage and error notification. The core of the communication system consists of five inter-related components as shown in Figure 3. These components are the Session Manager (SM), Buffer Manager (BM), Communication Object Class Library, Communication Object META Class, and CLIPS Interface Module. Collectively, these components provide the communication of objects among a dynamic set of heterogeneous clients.

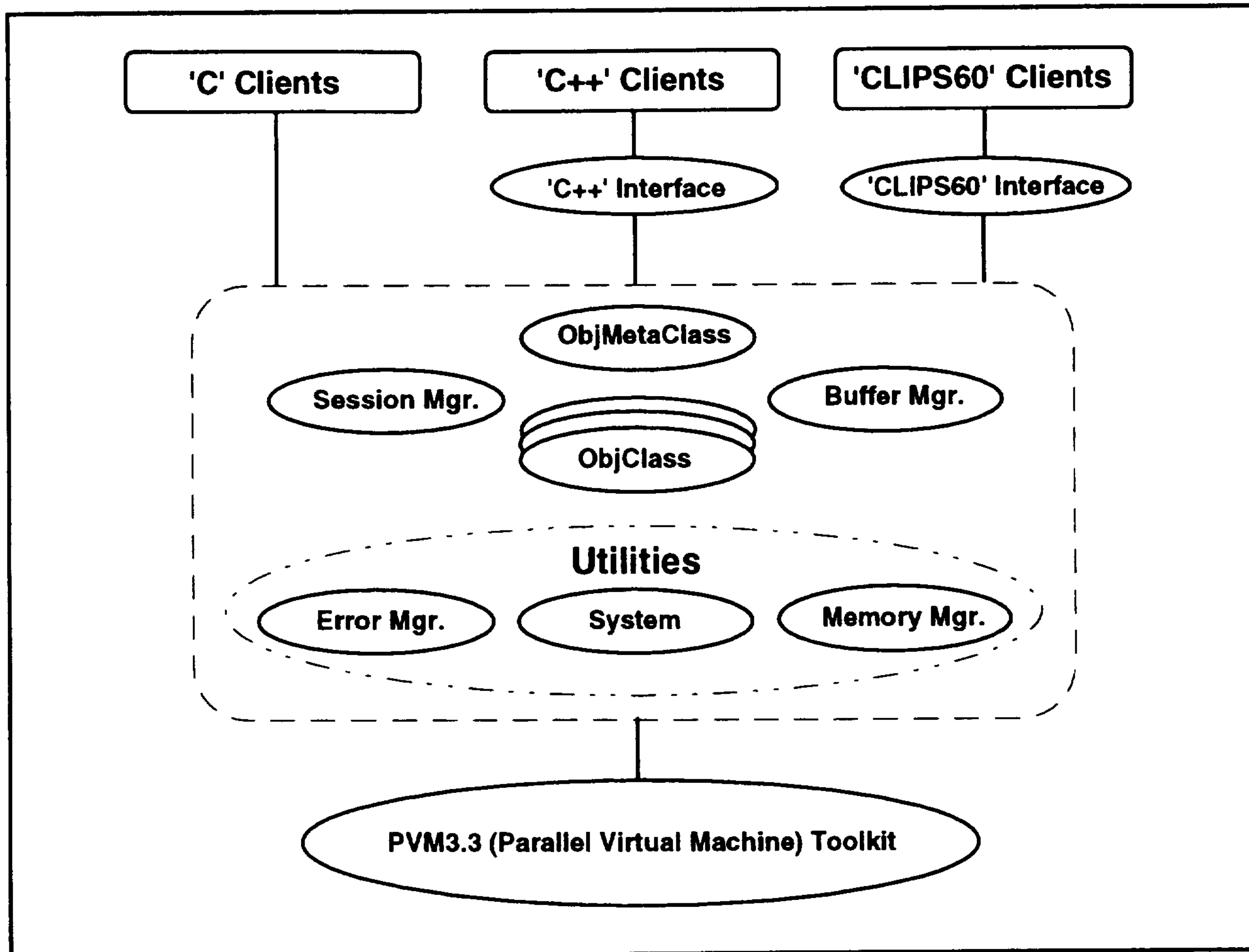

Figure 3. Communication System Architecture

Session Manager

The Session Manager (SM) allows clients to enter and exit application sessions. Within a specific session, clients can enter and exit any number of client groups. This functionality is particularly useful in sending messages to all members of a group. The SM can also invoke PVM to spawn additional application components utilizing a variety of configuration schemes. The facilities supported range from the specification of CPUs to host the spawned applications to an automatic selection of CPUs to use as hosts and balance the load across the virtual machine.

Buffer Manager

The buffer manager provides for the creation and manipulation of any number of fact buffers. Dynamic in nature, these buffers can be used to communicate several fact objects as a single atomic unit. That is, receiving CLIPS clients will accept and process the buffer's contents within

a single execution cycle. Clients are free to 'put' or 'get' facts into and out of buffers throughout the life of the buffer. Information as to the contents of a buffer can be obtained by querying the buffer directly.

Communication Object Class Library

The Communication Object Class Library contains methods which are specific to a particular data object. (These are the routines referred to in the Overview of CMS section above.) These methods include a set of constructors and destructors along with methods to 'pack' and 'unpack' an object component-by-component. This library includes additional methods to translate an object to and from the CLIPS environment and the C Object representation used with PVM. It is this class library which may require additions if new template forms are to be communicated.

Communication Object META Class

The motivation of the Communication Object META Class is twofold. The META Class combines common object class functionality into a single collection of META Class methods. That is, the highest level communication functions are exist as META Class methods, which freely accept any object defined in the Object Class Library without concern for its type. The same high-level META methods are invoked regardless of the type of objects in a communication buffer. They determine the actual type of object to be processed and then call the appropriate class method. This effectively allows clients to deal with a single set of methods independent of object type.

CLIPS Interface

Essentially acting as a client to the PVM system, this collection of modules extends the functional interface of the communication system described above to CLIPS processes. This interface provides CLIPS clients with the same functionality as their C and C^{++} counterparts. In an effort to preserve the syntactical style of the CLIPS assert command, several parsers were incorporated. Since the standard CLIPS external user functions interface does not support such functionality, some additional source code to the CLIPS distribution code was required. The following section provides a brief description of the functional interface presented to CLIPS clients.

CMS FUNCTIONS

ID_ObjRef SMConnect()

Connects the caller to a multi-agent session.

ARGUMENTS :STRING Name
LONG InitIdServerValue
LONG IdServerRange

RETURNS : A reference to the object representation of the caller or EM_ERROR_REF. Note, this object can be passed to other clients as a way to address the caller directly.

int SMDisconnect()

Disconnects the caller from a multi-agent session.

ARGUMENTS : void

RETURNS :EM_SUCCESS or EM_ERROR

int SMEnterGrp()

Enrolls the caller into the group <Group>.

ARGUMENTS : STRING Group

RETURNS : The caller's instance number within <Group> or EM_ERROR. Instance numbers start at 0 and increment upward.

int SMExitGrp()

Unenrolls the caller from the group, <Group>.

ARGUMENTS : void

RETURNS : EM_SUCCESS or EM_ERROR

int SMSpawn()

Spawns <NumCopies> copies of <Task> in accordance with the values of <How> and <where>.

ARGUMENTS : SYMBOL Task
INTEGER NumCopies
INTEGER How

<SM_DEFAULT - Target host is determined by the Session Manager. In this case <Where> should be NULL.
SM_HOST - Target host is determined by <Where>.
SM_ARCH - Target host is determined by the Session Manager. The target host's architecture will be of type <Where>.
SM_DEBUG - Target host is determined by the Session Manager. In this case <Where> should be NULL. All copies of <Task> will be run in the PVM debugger.
SM_TRACE - Target host is determined by the Session Manager. In this case <Where> should be NULL. All copies of <Task> will generate PVM trace data. >

SYMBOL Where
<Examples: "moby.cadrc.calpoly.edu" , or SUN4>

MORE ARGS ArgV
<Any # of args passed to each task upon startup.>

RETURNS : Actual number of spawned tasks or EM_ERROR.

int BMPutTmplt()

Places <Tmplt> into <BufRef>.

ARGUMENTS : BM_BufRef BufRef
DEFTEMPLATE Tmplt
RETURNS : EM_SUCCESS or EM_ERROR

int BMGrpSend()

Sends the contents of <BufRef> to each member of <Group>.

ARGUMENTS : STRING Group
BM_BufRef BufRef

RETURNS : EM_SUCCESS or EM_ERROR

int BMQueryReceive()

Formally processes incoming object groups. If no pending object groups exist, control is IMMEDIATELY returned to the caller.

ARGUMENTS : void
RETURNS : EM_SUCCESS or EM_ERROR

int SGrpSendIdentity()

Sends an identity to each member of a group.

ARGUMENTS : STRING Group
ID_ObjRef ObjRef
RETURNS : EM_SUCCESS or EM_ERROR

int SGrpSendTmplt()

Sends a template to each member of a group.

ARGUMENTS : STRING Group
Expression Tmplt
RETURNS : EM_SUCCESS or EM_ERROR

int SPrintIdentity()

Prints the object referenced by NIdentityRef> to stdout.

ARGUMENTS : ID_ObjRef IdentityRef
RETURNS : EM_SUCCESS or EM_ERROR

BM_BufRef BMCreate()

Creates a "send" buffer which encodes its contents according to <Encoding>.

ARGUMENTS :INTEGER Encoding
<BM_DEFAULT - Data for heterogeneous networks
BM_RAW - No encoding takes place
BM_IN_PLACE - Same as BM_DEFAULT except data is not copied into the buffer until it is sent. >
RETURNS : A reference to buffer or EM_ERROR_REF

int BMDestroy()

Destroys the buffer referenced by <BufRef>.

ARGUMENTS : BM_BufRef BufRef
RETURNS : EM_SUCCESS or EM_ERROR

int BMSend()

Sends contents of <BufRef> to <IdentityRef>. The contents of <BufRef> is left unaltered.

ARGUMENTS :ID_ObjRef IdentityRef
BM_BufRef BufRef
RETURNS : EM_SUCCESS or EM_ERROR

int BMReceive()

Formally processes incoming object groups. If no pending object groups exist, caller is put to sleep until such an event occurs. Received objects will be placed into the caller's fact-list in their appropriate form.

ARGUMENTS : void
RETURNS : EM_SUCCESS or EM_ERROR

int SSendIdentity()

Sends an identity to another client.

ARGUMENTS :ID_ObjRef DstIdentityRef
ID_ObjRef ObjRef
RETURNS : EM_SUCCESS or EM_ERROR

int SSendTmplt()

Sends a template to another client.

ARGUMENTS :ID_ObjRef DstIdentityRef
Expression Tmplt
RETURNS : EM_SUCCESS or EM_ERROR

Figure 4. CMS Functions

A CMS EXAMPLE

The following example illustrates a CLIPS knowledge base intended to communicate **VALUE_FRAME** templates with CMS. The example consists of four CLIPS constructs: one deftemplate and three defrules. The first step of the example is to define a **VALUE_FRAME** template having four slots. The client registers itself in the session via the **CONNECT_ME** rule. This rule also enters the client into a group. The rule then broadcasts the client's identity to all other members of its group. Incoming identifications are processed by the **RECEIVE_IDENTITY** rule. After receiving another client's identification, several **VALUE_FRAME** facts are placed into a buffer and sent back to the client.

Incoming **VALUE_FRAME** facts are processed by the **PROCESS_VALUE_FRAME** rule. Finally, when there are no more rules on the agenda, the client goes into a blocking receive state via the execution of the **RECEIVE** rule. CLIPS clients receive facts in two distinct manners. In the first case, the communication system is queried at the end of each execution cycle for pending messages. The second method by which a client can receive messages is through an explicit call to BMReceive. The functionality of this call is identical to the implicit method, with the exception that the caller will be put to sleep until a pending message exists. In either case, incoming facts are processed transparently to the client and produce immediate modifications of the fact-list.

```
(deftemplate VALUE_FRAME
    ( slot    Frame    )
    ( slot    Instance )
    ( slot    Slot     )
    ( slot    Value    )

)


(defrule RECEIVE_IDENTITY
    ( BM_DEFAULT ?Mode )
    (    SESSION-MEMBER ?Name
         CAN-BE-REFERENCED-BY ?Identity
    )
=>
    ; Send our new friend some templates
    ( bind ?BufRef ( BMCreate ?Mode ) )

    ( BMPutTmplt ?BufRef ( VALUE_FRAME
                            ( Frame  Frame1 )
                            ( Instance   Instance1 )
                            ( Slot   Slot1 )
                            ( Value  1234 )
                         )
    )
    ( BMPutTmplt ?BufRef ( VALUE_FRAME
                            ( Frame  Frame2 )
                            ( Instance   Instance2 )
                            ( Slot   Slot2 )
                            ( Value  56.789 )
                         )
    )
    ( BMPutTmplt ?BufRef ( VALUE_FRAME
                            ( Frame  Frame3 )
                            ( Instance   Instance3 )
                            ( Slot   Slot3 )
```

```
(defrule CONNECT_ME
    ( initial-fact )
=>
    ( bind ?MyName "Access" )
    ( bind ?MyGrp "AgentGrp" )

    ; Connect to the session
    ( bind ?MyId ( SMConnect ?MyName 200 200) )

    ; Join a group
    ( SMEnterGrp ?MyGrp )

    ; Send my identity to members of my group
    ( SGrpSendIdentity ?MyGrp ?MyId)


    ; Assert the "receive" control fact

    ( assert( RECEIVE ) )
)


    (defrule PROCESS_VALUE_FRAMES
        ?TMPLT <- ( VALUE_FRAME
                        ( Frame ?Frame )
                        ( Instance ?Instance )
                        ( Slot ?Slot )
                        ( Value ?Value )
                  )
     =>
            ; Process the template

              . . .

            ( retract ?TMPLT )
    )
```

```
                    ( Value  "How are you ?" )
                )                                   (defrule RECEIVE
    )                                                   (declare (salience -10000 ) )
    ; Send the buffer to our new friend?                ?REC <- ( RECEIVE )
    ( BMSend ?Identity ?BufRef )                    =>
                                                        ( BMReceive )
    ; Destroy the buffer                                ( retract ?REC )
    ( BMDestroy ?BufRef )                               ( assert( RECEIVE ) )
)                                                   )
```

Figure 5. A CMS Sample

CONCLUSION

PVM and CLIPS both provide free source code systems that are well maintained by developers and a sizable number of users. Relative few source code changes are necessary to either system in order to build a reliable and robust platform that will support distributed computing in a heterogeneous environment of CPUs operating under UNIX. The CMS system described in this paper provides the CLIPS interface code and some parsing code sufficient to enable efficient use of PVM facilities and communication of CLIPS facts and templates among C, C++, and CLIPS processes within a PVM virtual machine. Even more efficient communication can be obtained through enhancements to the PVM source code that can provide more efficient allocation of memory and reuse of PVM message buffers in certain applications.

NOTES

Information on PVM is best obtained by anonymous ftp from: netlib2.cs.utk.edu
Shar and tar packages are available from the same source.

The authors are currently using the CMS system in applications that involve multiple CLIPS expert systems in sophisticated interactive user interface settings. It is expected that the basic CMS code will become available in the Spring, 1995. Inquiries via e-mail are preferred.

BIBLIOGRAPHY

1. Riley, Gary, B. Donnell et. al., "CLIPS Reference Manual," JSC-25012, Lyndon B. Johnson Space Center, Houston, Texas, June, 1993.

2. Pohl, Jens, A. Chapman, L.Chirica, R. Howell, and L. Myers, "Implementation Strategies for a Prototype ICADS Working Model," CADRU-02-88, CAD Research Unit, Design Institute, School of Architecture and Design, Cal Poly, San Luis Obispo, California, June, 1988.

3. Myers, Leonard and J. Pohl, "ICADS Expert Design Advisor: An Aid to Reflective Thinking," Knowledge-Based Systems, London, Vol. 5, No. 1, March, 1992, pp. 41-54.

4. Pohl, Jens and L. Myers, "A Distributed Cooperative Model for Architectural Design," Automation In Construction, Amsterdam, 3, 1994, pp. 177-185.

6. Geist, Al, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam, "PVM 3 User's Guide and Reference Manual," ORNL/TM-12187, Oak Ridge National Laboratory, Oak Ridge . Tennessee, May, 1993.