SRC TR 87-25

Architecture of MRMS Simulation Distributing Processes

by

V. Sinha

Architecture of MRMS Simulation: Distributing Processes

Velu Sinha *
Systems Research Center
The University of Maryland
College Park, MD 20742

January 1, 1987

Abstract

As described in [3], the Mobile Remote Manipulator System Simulator is based on an interconnected network of heterogenous computers. The simulation is divided into modules which run concurrently on multiple computers. Modules are designed so that they perform a specific task which can be used in a variety of simulations. As each of these modules is built, it is necessary to provide a method for intermachine / intermodule communication. This paper describes various methods which can be used for this type of communication, and also describes various standard data formats which are used to get data from module to module in the MRMS Simulator.

1 Construction of Modules

A module can be built out of any algorithm which has a well defined input data stream and output data stream. The location of the module is dependent on the algorithm: e.g. for mathematically complex tasks, a computer with a FPU makes an ideal host, while for a graphically intense task, a workstation with graphics primitives is the host of choice.

To convert an algorithm into a module it is necessary to do the following:

1. Decide on a standard data format for input and output streams [1].
2. Add stream based i/o.

*This work was supported in part by NSF Grant #85-00108, AFOSR-URI Grant #AFOSR-87-0073, and by a Design Project Grant from NASA through the Universities Space Research Association.

[1] Standard data formats are discussed in the Appendix.

3. Add network interface for streams.

4. Add exception handling into stream based system.

Once these steps are complete, the module can be integrated into the simulation system. One factor which should not be overlooked when building a system of this sort is how and where a set of modules which run a simulation are started, and how they give their output. In the MRMS case this was quite well defined, the simulation was started from the Iris 2400 workstation, the results were shown on the Iris 2400 workstation, and the simulation parameters were controlled from the workstation. Such a circular arrangement is possible as the Iris is serving two distinct functions in the simulation: It is acting as both the control panel for the MRMS system, as well as the MRMS itself.

2 Structure of the Algorithm

As mentioned above, it is necessary to add four basic steps to turn an algorithm into a module. However, these steps can be more easily added to some algorithms than others. It is easier to convert an algorithm into a module if the following guidelines are followed while implementing the algorithm:

- Write the algorithm so that it expects input and output from data streams. Do not query the "user" for input in the module, use a well defined and compact input/output format, preferably conforming to one of the standards in the appendix. Allow for *stdin* and *stdout* to be specified as the input/output source.

- Write the algorithm with a "forever" loop, which allows for the function of the algorithm to be called over and over again. Allow for the algorithm to terminate elegantly upon the proper termination sequence.

- Provide a mechanism to reset the algorithm in mid-input. This should return the algorithm to a pristine state so that a dataset with higher priority can be processed. Also allow the various parameters (constants) which the algorithm uses to be modified using an appropriate command sequence.

- Use the C programming language wherever possible. This will allow the algorithm to be moved from machine to machine most easily. If you are using a Lisp Machine, please refer to [2] for more information.

If the above guidelines are followed the task of converting the algorithm into a module will be simplified.

In order to assist users in preparing modules, a sample module written in C is given in the appendix.

3 Inter-Process / Inter-Machine Communication

As most of the modules to date have been written under UNIX [2] heavy use has been made of the UNIX IPC (Inter-Process Communication) facilities[1]. One advantage of using these facilities is that under UNIX no distinction needs to be made between local processes and remote processes, where a process runs (or in this case, which machine a module is located on) can be completely transparent to the user. As a matter of fact this can even allow for a simple fault tolerant mechanism to be built into the system — if a host goes down, then the modules which were running on that host can be started up automatically on other hosts. Of course, the performance of the system will go do in situations like this, but the user will not need to be aware of such things.

When first implementing the IPC code in the simulation modules, there was a very tight link between the IPC routines and the modules themselves. However, upon further study is was decided that each module should be started up along with a sister process which handles all of its outgoing communication routing. Each process is then started up in one of two ways: Manually, from a terminal, mostly for testing purposes, or automatically by the Simulation Server which runs on each machine.

Currently all modules are started manually, however, a simulation server has been implemented, and is currently being tested.

3.1 Simulation Server

Each host participating in the simulation has a "simulation server." The simulation server listens to the network for connections from other systems, or from other modules on the same system. Upon receiving a connection, the server routes the data to the appropriate module. If the module is not currently being executed, the server starts the module.

A module sends its output to another module through its sister IPC process. A sister process is used because it allows the modules to be linked and tested most easily: When all of the modules reside on one host they are started up with simple UNIX pipes between them. The sister processes interact with the modules through pipes also, but have the capability for opening connections to simulation servers on other hosts.

It is very important to note that there is a fixed direction in which all messages flow. Once modules are linked together, messages flow only in one direction. At the moment, it is not possible to dynamically add a new module to the simulation — if a module may even possibly be used, it should be specified when the simulation is started. The module can be turned on or off by use of a command such as the ones described in the next section.

[2] UNIX is a trademark of AT&T Bell Laboratories.

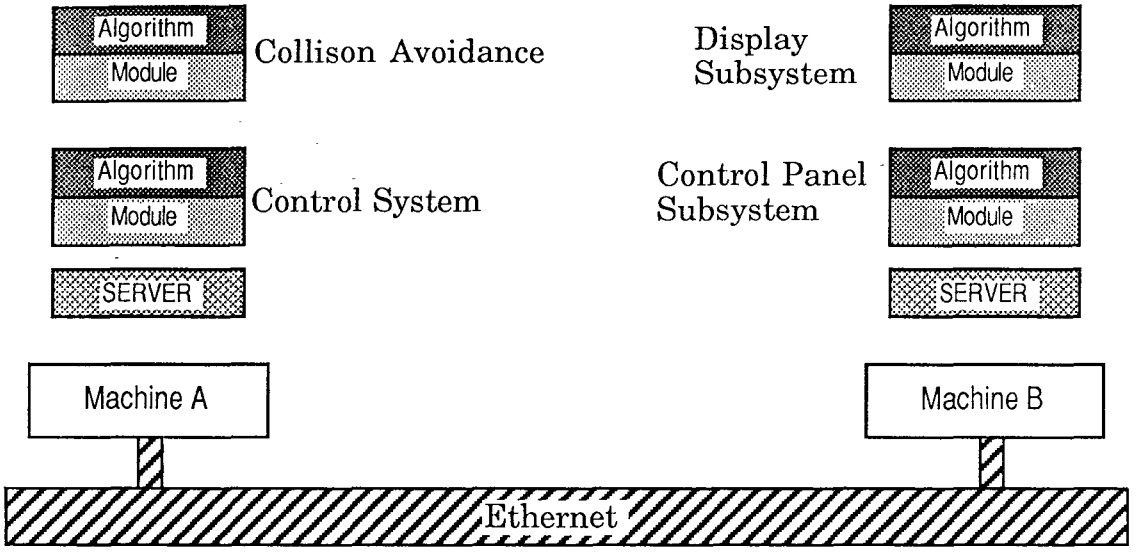

Figure 1: Hardware and Software Architecture of Simulation System

Figure 1 shows both the hardware and software architecture of the simulation system.

4 Appendix A: Simulation Data Formats

There are many different kinds of data which are currently sent from module to module, and there will be many more. The following data format provides a standard which can be easily extended as new data formats are needed. Each message has the following mandatory fields:

```
Sender Recipient Command [Variables]
```

Each of SENDER, RECIPIENT, and COMMAND is an integer between 0 and $2^{16} - 1$. Every module is assigned a unique number.

4.1 Initialization Data

The first message sent to any module is a reset instruction. This instruction has the following format:

```
0 0 0 Xmin Ymin Zmin Xmax Ymax Zmax
```

The two leading zeros imply that it is a broadcast message (from 0, or the simulation controller, to 0, which is a pseudonym for everyone), and the third zero implies that it is a reset command.

The reset command has six mandatory variables, which define the two corners (FLL and the RUR corners) of a box which contains the simulation universe.

If a module ever receives an initialization command, the algorithm should be reset to its pristine state.

4.2 Operational Data

The following data formats are used when sending operating commands and the appropriate data through the system. In the table below the Sender and Recipient fields are omitted, only the Command and variable fields are given.

N.B. The command numbers are specific only to a module, though for ease of programming it may be simplest to assign unique numbers to each command, regardless of module. The commands below apply solely to the display module.

```
1   ViewX ViewY ViewZ TowardsX TowardsY TowardsZ
2   FOV
10  BasePos BaseRot Jnt1 Jnt2 Jnt3
11  Obs_Num ObsFflX ObsFflY ObsFflZ ObsRurX ObsRurY ObsRurZ
12  Object_Number ObjX ObjY ObjZ
13  Object_Number_To_Attach_Viewpoint_To
```

5 Appendix B: Sample Module

The following is the code for a simple linear interpolator used in lieu of a control system in the original (circa May 86) MRMS demonstration.

```
#ifndef lint
static char sccsid[] = "@(#)smooth.c       U of Md *SRC* VS 21-May-1986";
#endif
#include <stdio.h>
#include "mrms.h"

#define ME       CONTROL_SYS       /*Control Sys Module Number*/

/* This module takes the place of a control system by performing simple
*  linear interpolation between specified positions of the MRMS.
*
*       Velu Sinha
*/

main()
{
    float v[5];
    float ov[5];
    float d1, d2;
    int flag;
    int x;
    float sign, step;
    float diff;
    int From, To, Command;
    int havept=0;

    while(scanf("%d %d %d",&From, &To, &Command) == 3){
/*
*  Read From, To, and Command fields...
*/

       if(To && To != ME){
/*
*  If the message is not for us, and is not 0 (the reset message),
then
*  rebroadcast it.
*/

           printf("%d %d %d ", From, To, Command);
```

```
          while((x=getchar())!='\n') putchar(x);
          continue;
/*
*  Go and wait for the next message...
*/

        }  /* closing the if(To && ... statement */
/*
*  OK, well, this message IS for us...
*/
       switch(Command){
/*
*  Let us figure out what command it is...
*/
          case 0:
/*
*  RESET...
*/
             d1=d2=v[0]=v[1]=v[2]=v[3]=v[4] = 0.0;
             flag = 0;
             havept = 0;
/*
*  Pass the message on...
*/
               printf("%d %d %d %f %f %f %d %f %f %f %f\n",
                    ME, DISPLAY, MOVE_MRMS,
                       d1, d2, v[0], flag, v[1], v[2], v[3], v[4]);
               break;
/*
*  This one means that we interpolate...
*/
          case 10:
              scanf("%f %f %f %d %f %f %f %f\n",
                      &d1, &d2, &v[0], &flag, &v[1], &v[2], &v[3],
&v[4]);
/*
*  We don't know how to handle this, yet...
*/
               if(flag == -1){
                       printf("%d %d %d 0. 0. 0. -1 0. 0. 0. 0.\n",
                                       ME, DISPLAY, MOVE_MRMS);
                       exit(0);
               }  /* if */
/*
```

```
*  Save a copy for interpolation purposes...
*/

               for(x=0; x<5; x++)
                       ov[x]=v[x];
/*
*  If I have nothing to interpolate with, then pass this on.
*/
              if(!havept){
                       printf("%d %d %d %f %f %f %d %f %f %f %f\n",
                            ME, DISPLAY, MOVE_MRMS,
                               d1, d2, v[0], flag, v[1], v[2], v[3],
v[4]);
                     havept=1;
                     break;
              }  /* if */

/*
*  Otherwise, Look for the first value which has changed, that is the
value
*  to be interpolated.
*/
                for(x=0; x<5; x++)
                        if(abs (v[x] - ov[x]) > .02) break;
/*
*  If this happens (ie, if this new point is far enough away to make
*  a difference...
*/
                if(x != 5) {
                        diff = v[x] - ov[x];
                        if(x == 0)
                                step = 1.0;
                        else
                                step = 0.2;

                        sign=1;

                        if(diff < 0) step*=(sign = (-1));

/*
*  ... calculate interval, and interpolate...
*/
                        for(; sign*ov[x]<=sign*v[x]; ov[x] += step)
```

```
                                printf("%d %d %d %f %f %f %d %f %f %f
%f\n",
                                    ME, DISPLAY, MOVE_RMS,
                                         d1, d2, ov[0], flag, ov[1],
                                         ov[2], ov[3], ov[4]);
                } /* if */

        } /* switch */
    } /* while */
} /* main */
```

6 Appendix C: Users guide to starting a distributed simulation

To start a MRMS simulation, login to the Iris, and issue a command of the following format...

```
simulate MACH {data source} [MACH module ...] | dsply_prog
```

or

```
rsh MACH data source | rsh MACH module | rsh MACH module | dsply_prog
```

Where MACH is any one of the valid hostnames, and data source is a program which creates the data (can be of the form *cat filename* if the data is precomputed and stored in a file). Module is the location of the module in the filesystem of the specified host. There may be any number of MACH module statements.

References

[1] Samuel J. Leffler, Robert S. Fabry, and William N. Joy. *A 4.3BSD Interprocess Communication Primer.* Computer Systems Research Group, Department of EE & CS, University of California, Berkeley, December 1985. Available on ENEEVAX in /usr/doc/ipc.

[2] Velu Sinha. *Architecture of MRMS Simulation: Lisp Machine Interface.* Intelligent Servosystem Laboratory, Systems Research Center, The University of Maryland, College Park, MD, 1987. In Preperation.

[3] Velu Sinha. *The Mobile Remote Manipulator System Simulator.* Intelligent Servosystem Laboratory, Systems Research Center, The University of Maryland, College Park, MD, December 1986.