*Proceedings of the 1990 Winter Simulation Conference*
Osman Balci, Randall P. Sadowski, Richard E. Nance (eds.)

# DISTRIBUTED SIMULATION: NO SPECIAL TOOLS REQUIRED

Frank Paterra
C. Michael Overstreet
Kurt Maly

Department of Computer Science
Old Dominion University
Norfolk, Virginia 23529-3915

## ABSTRACT

In this paper the authors present a toolkit of C language functions that can be linked with SIMSCRIPT programs to provide the data communication primitives necessary for distributed simulation. The authors' test case is discussed and some timing data are presented. Additionally metrics used to determine the applicability of the server model decomposition for particular simulations are discussed.

## 1. WHY DISTRIBUTE SIMULATIONS

Computer simulations are often computationally intense tasks requiring long runs in order to obtain useful results. The runtime requirements of a simulation model can be a problem both during model development and validation and while performing production runs of the simulation.

The development of computer models to simulate real world objects is a well understood problem and number of tools exist to aid the model developer [Gimarc 1989]. Often times, the initial runs of a simulation model provide more questions than answers and the focus of study is changed. This results in an evolutionary process for model development, with refinements directed at different attributes as the object or its environment becomes better understood. These refinements often increase the complexity of the model.

As the model is evolving, many runs may be needed to better understand the object and to verify the correctness of the simulation. The runtime requirements of complex models can greatly increase the time needed to develop and verify a model.

Once a model has evolved to the point that production runs are being made, runtime requirements again come into play. Often the output from each run may only provide a single data point for a graph. A study may then require multiple runs of the simulation, differing only in input values.

Complementary to the problem of long, computationally intensive, runtimes is the fact that many times other computers are sitting idle and can provide basically free processor cycles to the simulation. In an effort to utilize some of these free cycles, much research has gone into developing algorithms for performing a single simulation run on a number of loosely coupled, cooperating processors.

Distributing a simulation program among cooperating processors introduces some difficult problems. Principal among these are the identification of an effective decomposition of the simulation program, and enforcing synchronization among processors to insure that the simulation is being executed correctly. The use of tightly coupled functions and dependence on shared data, common in simulation programs, makes these problems acute to distributed simulation.

A comprehensive treatment of the processor synchronization and model decomposition problems can be found in [Fujimoto 1989; Jefferson 1987; Cota and Sargent 1986]. The problem of processor synchronization is more easily solved in very tightly coupled processors that support very high speed communication, though it is still difficult [Kaudel 1987; Jones et al. 1989]. It is not our intention to address these problems in this paper, but rather to select an effective decomposition and synchronization scheme that will be used to demonstrate distributed simulation using our communication toolkit and standard simulation and operating system tools.

High performance scientific workstations sharing a local area network (LAN) are becoming common. Since shared memory is not available and message passing among workstations in the network is slow, a decomposition of the simulation task is only likely to to be effective if messages are infrequently passed among workstations. It is up to the modeler/programmer to find a effective decompositions gien these constraints. This toolkit has been developed with this environment in mind.

## 2. MODEL DECOMPOSITION

The model decomposition most easily supported by the tools described in this paper is to distribute some special types of model components, here called servers and receivers, on different machines. The term server is borrowed from object oriented design: a component is a server submodel if it can be represented as only sending to other model components.

This decomposition can be thought of as a collection of data servers and and receivers with no cycles. With no cycles synchronization becomes easier and the problems with deadlock such as that described in [Bagrodia et al. 1987] are avoided. This is an easy and usually useful decomposition for complex, tightly coupled models, because the extensive data interaction among the model's parts are not interfered with. Other, potential more effective decompositions are outside the scope of this paper.

These tools are based on a producer/consumer approach. Each consumer keeps a local inventory of data objects which have been created by a producer. When the consumer's inventory gets low, it "reorders" a new batch of data objects from the appropriated producer so that new objects should arrive before current supplies are exhausted. In addition, each producer prepares an inventory of data objects so that it should be able to respond immediately to resupply requests.

The goal with this approach is to allow computational tasks to be performed concurrently on different workstations but to require only infrequent communication among them. The approach also recognizes that in most LANs, the overhead and time delays of sending a small amount of data from one workstation to another is almost the same (up to the limits of the maximum packet size allowed) of sending a large amount of data.

If data provided by the servers require considerable computation, significant parallelism can result since the required computation can be distributed to other workstations. If the model is decomposed in such a way that no data cycles are present, no possibility of deadlock exists and processor synchronization is particularly simple.

The type of model decomposition required to benefit from these tools is probably not achievable for all models, but our experience is that the decomposition is possible frequently enough to make the approach and the tools worthwhile.

## 3. DISTRIBUTED SIMULATION WITH STANDARD TOOLS

In this paper we describe a toolkit of functions that allows distributed simulation to be carried out in a loosely coupled, general purpose, workstation environment without the use of special purpose operating systems, programming languages, or hardware.

The particular environment for which this software was developed contains a collection of Sun workstation computers connected via an ethernet LAN. These are very loosely coupled UNIX workstations with no shared memory and only communicate via a shared bus (the ethernet LAN). SIMSCRIPT was selected as the simulation language because of its wide use in the simulation community. All processors cooperating in the simulation run programs written in SIMSCRIPT and call external functions for processor communication.

The processor communication functions are provided via the UNIX Interprocess Communication (IPC) functions described in the Unix Programmer's Manual. These functions are standard with the BSD UNIX operating system and allow communication among processes both within the same computer and those residing on separate computers. Because the IPC functions are designed to provide communication among a large number of varying processor types, a significant amount of overhead is inherent with data communication. This could be reduced by writing replacement functions that only provide for the needs of this simulation decomposition, however a design goal was to use as little custom software as possible.

## 4. THE TOOLKIT

The toolkit consists of a collection of functions, written in the C language and linkable with SIMSCRIPT programs. The basic functions provided by the toolkit are interprocess data communications and sufficient processor synchronization to allow a simulation to be decomposed into a collection of data servers and receivers.

To use these tools, one must first determine what in their model can be thought of as a source or generator of precomputable data objects. In order for an object to be precomputed, no information about current simulation state or access to variables not local to the generator should be required. The most obvious precomputable object is random numbers, however, more complex objects may be precomputed based on the simulation model at hand. Once the data sources have been identified, the simulation is written as usual, except that the identified source objects are written as separate SIMSCRIPT programs. This results in the simulation being implemented as data generator programs and a simulation program. Two C language functions must be called by both the simulation program and the generator program to install the communications interrupt handler and to identify each of the generators participating in the simulation. Each of the SIMSCRIPT programs must also contain a function that the C routine calls to transfer generated data to and from SIMSCRIPT variables. The SIMSCRIPT and C functions are described below.

### C Functions

- inst_int(host,mode)
  - char *host - The name of the host running the receiver program
  - char *mode - Must equal "server" or "receiver" for the server and receiver programs respectively.

  This is a C routine that is called by both the receiver and server programs. Called only once, and before any of the link server function described below, this function opens a socket for reading, installs the communications interrupt handler, and initializes the variables used to maintain the server information.

- link_server(server,host,mode)
  - char *server - The name of the server being identified
  - char *host - The name of the host where the server resides
  - char *mode - Must equal "server" or "receiver" for the server and receiver programs respectively.

  This C routine is called by both the receiver and server processes to identify the services that are being used in this simulation. The function creates a record of the identified server's information, opens a sending socket for the server, initializes the list of messages to that server as null, and adds the server to the list of participating servers.

- request(service,command)
  - char *service - The name of the service being requested
  - int command - A command to be sent to the server. This command integer is not examined by the toolkit; it is completely definable by the model and server developers.

  This C routine is called by the simulation module to request more data items from a server. After the request is sent, control is return to the simulation software. When the requested data are received, the simulation code will be interrupted and the toolkit makes a call to a user provided accept_data routine, described below.

**SIMSCRIPT Functions**

- accept_data given service, data, length
  - service - Text variable containing the name of the service supplying the data. This field is used to route data to the correct inventory.
  - data - Memory for objects created. This memory space will be formatted by the server to hold the data in the correct SIMSCRIPT format.
  - length - The length in bytes of the data area.

  The accept_data function is called by the C routine that performs the socket reads when new data arrives. Because the arrival of data causes an interrupt to be serviced and this function is called during that interrupt, the code may be executed at any time; this function has an impact on the simulator's use of pointers or indices to the inventory of data.
- fill_request given service, data, yielding length
  - service - Text variable containing the name of the service being requested.
  - data - Memory for objects being created. This is unformatted memory and can be interpreted and filled according to the objects being requested.
  - length - Integer variable returning the length in bytes of the data to be supplied.

  This SIMSCRIPT function is the server complement to the accept data function. When a request for objects is received by the communications handler, this function is called to fill the request. As before, because the communications are interrupt driven, this function can be called at any time.

After the receiver code has been moved to a separate program, additional SIMSCRIPT code is needed in the receiver program to manage the remotely generated data. This additional code keeps track of the available inventories of remotely generated data, placing requests for additional data when the local inventory falls below some threshold. How this threshold is calculated is discussed in a later section.

The receiver program may itself be a data source. An example is the generation of simulation data that are sent to additional programs that provide statistical analysis and summary reports or to other servers for graphical display.

## 5. TIMING DATA AND MODEL DECOMPOSITION CONSIDERATIONS

Message passing overhead must be considered when designing any type of distributed processing. To decide if any speedups can be expected for a simulation using the server model decomposition, some analysis for message passing times verse local computational costs should be performed. The following definitions are used to perform this analysis.

- CIO - Overhead induced by servicing a communications interrupt. This includes the time required to transfer data from the communications buffer to a SIMSCRIPT variable.
- CSO - Overhead induced by actively sending a message to another server.
- CTT - Time for a command message to travel between two hosts.
- DTT - Time for a data message to travel between two hosts.
- MST - Minimum time required to supply data objects.
- OS - Order size. The number of items shipped in each order.
- ECR - Expected consumption rate for generated items.
- REQ - Time required to send a request to the server.
- REC - Time required to send objects to the receiver.
- LCT - Local computational time required to generate one data object.
- RGT - Remote generation time. The time required by the server to generate the values. This value is determined by OS and LCT.
- TBO - Time between orders.

Actively sending a request for more data objects incurs the expense of building a message and placing it on the communications medium. Receiving a message requires that the communications interupt be serviced and the received data be transfered from the communications meduim to a buffer. With this in mind we can define REQ and REC to be

$$REQ = CSO + CTT \tag{1}$$

$$REC = CIO + DTT \tag{2}$$

To determine values for MST, the values of REC and REQ must be considered as well as how much of the requested data can be precomputed between request orders. The total time required to compute data objecs to fill an order is $LCT * OS$. Because the server computers can work between orders, some of the computation may be done in the time between the orders. The amount that can be precomputed depends on the $LCT$ and $TBO$, described by the relationship

$$(LCT * \max((OS - TBO/LCT), 0) \tag{3}$$

Using this, $MST$ can be described by equation 4. This is the equation used when constructing table 4.

$$MST = REQ + REC + (LCT * \max((OS - TBO/LCT), 0) \tag{4}$$

Clearly the relationships below must hold or it will always be faster to compute the objects locally.

$$OS \geq MST * ECR \tag{5}$$

$$OS * LCT > CIO + CSO \tag{6}$$

In many cases, unless LCT (the required to computing the data locally) is significant, OS must be large to make this approach feasible. Practically speaking, since for most simulations the actual consumption rate can vary, $OS * LCT$ should be significantly larger than $CIO + CSO$.

In order to assist in determining the potential effectiveness of using this approach for distributed simulation, some timing data were collected for LCT, CIO, CSO, DTT, and CTT. The

variable ECR is model dependent and, with the other variables fixed, OS can be determined.

Timing data for message passing among Sun workstations connected via ethernet networks was collected. The times required for message passing are not significantly affected by message length as long as messages are less than the maximum packet length for the ethernet (1500 bytes). Messages were passed between processors that resided on the same physical network and those on separate networks, connected via a bridge. As can be seen below, messages that had to go across bridges took twice as long as those that stayed on a single network. All data were collected when the network was lightly loaded. Tables 1 and 2 show the collected communication times for command and data packets respectively. Table 3 gives the average message times and throughput rates. Times are given in seconds/byte and throughput is given in bytes/second. These values are need to compute useful order sizes and were used to construct table 4. This table is discussed below.

**Table 1.** Command Packet Times (100 Byte Packets)

| Number of Packets | Single Net (seconds) | Bridged Nets (seconds) |
|---|---|---|
| 100 | 1 | 3 |
| 500 | 5 | 9 |
| 1,000 | 9 | 23 |
| 5,000 | 45 | 112 |
| 10,000 | 87 | 206 |
| 50,000 | 435 | 944 |

**Table 2.** Data Packet Times (1000 Byte Packets)

| Number of Packets | Single Net (seconds) | Bridged Nets (seconds) |
|---|---|---|
| 100 | 2 | 3 |
| 500 | 8 | 20 |
| 1000 | 16 | 33 |
| 5,000 | 79 | 171 |
| 10,000 | 156 | 332 |
| 50,000 | 793 | 1,721 |

**Table 3.** Average Times for Command and Data Packets

| Packet Size | Average Time (sec.) | | Throughput (bytes/sec) | |
|---|---|---|---|---|
| | Intra-net | Inter-net | Intra-net | Inter-net |
| 100 | 0.000087 | 0.000194 | 11,400 | 5,100 |
| 1,000 | 0.000015 | 0.000034 | 63,000 | 29,000 |

To obtain values for the variables LCT, CSO, and CIO, the UNIX *prof* command was used. This is a standard UNIX tool that profiles executable code and generates reports on number of times each function is called, time spent during each call, and total time spent in the function during program execution. For more information on the prof command see the Unix User's Manual.

## 6. EXAMPLE

As an example, consider the generation of normally distributed random numbers. The machines used are Sun 3/60 workstations with 8 megabytes of memory. LCT was found to be 0.1 ms; the CSO and CIO were both 0.025 ms. Table 4 shows values for OS with corresponding ECR values.

**Table 4.** Example Performance Data

| ECR (minute) | OS | Orders | RGT (seconds) | TBO (seconds) | MST (seconds) |
|---|---|---|---|---|---|
| 100,000 | 50 | 2,000 | 0.005 | 0.030 | 0.014 |
| | 100 | 1,000 | 0.010 | 0.060 | 0.020 |
| | 500 | 200 | 0.050 | 0.300 | 0.068 |
| | 1,000 | 100 | 0.100 | 0.600 | 0.128 |
| | 5,000 | 20 | 0.500 | 3.000 | 0.608 |
| | 10,000 | 10 | 1.000 | 6.000 | 1.208 |
| 200,000 | 50 | 4,000 | 0.005 | 0.015 | 0.014 |
| | 100 | 2,000 | 0.010 | 0.030 | 0.020 |
| | 500 | 400 | 0.050 | 0.150 | 0.068 |
| | 1,000 | 200 | 0.100 | 0.300 | 0.128 |
| | 5,000 | 40 | 0.500 | 1.500 | 0.608 |
| | 10,000 | 20 | 1.000 | 3.000 | 1.208 |
| 300,000 | 50 | 6,000 | 0.005 | 0.010 | 0.014 |
| | 100 | 3,000 | 0.010 | 0.020 | 0.020 |
| | 500 | 600 | 0.050 | 0.100 | 0.068 |
| | 1,000 | 300 | 0.100 | 0.200 | 0.128 |
| | 5,000 | 60 | 0.500 | 1.000 | 0.608 |
| | 10,000 | 30 | 1.000 | 2.000 | 1.208 |
| 400,000 | 50 | 8,000 | 0.005 | 0.007 | 0.014 |
| | 100 | 4,000 | 0.010 | 0.015 | 0.020 |
| | 500 | 800 | 0.050 | 0.075 | 0.068 |
| | 1,000 | 400 | 0.100 | 0.150 | 0.128 |
| | 5,000 | 80 | 0.500 | 0.750 | 0.608 |
| | 10,000 | 40 | 1.000 | 1.500 | 1.208 |
| 500,000 | 50 | 10,000 | 0.005 | 0.006 | 0.014 |
| | 100 | 5,000 | 0.010 | 0.012 | 0.020 |
| | 500 | 1,000 | 0.050 | 0.060 | 0.068 |
| | 1,000 | 500 | 0.100 | 0.120 | 0.128 |
| | 5,000 | 100 | 0.500 | 0.600 | 0.608 |
| | 10,000 | 50 | 1.000 | 1.200 | 1.208 |
| 600,000 | 50 | 12,000 | 0.005 | 0.005 | 0.014 |
| | 100 | 6,000 | 0.010 | 0.010 | 0.020 |
| | 500 | 1,200 | 0.050 | 0.050 | 0.068 |
| | 1,000 | 600 | 0.100 | 0.100 | 0.128 |
| | 5,000 | 120 | 0.500 | 0.500 | 0.608 |
| | 10,000 | 60 | 1.000 | 1.000 | 1.208 |

Two points can be made from table 4. First, when the order size is small, the communication overhead for transferring objects from the server to the receiver becomes significant. In table 4, when the ECR was 400,000 and the OS was less than 500, the TBO was greater than the MST. This means that the consumer had to wait for the producer. Even in this case a speedup is possible because the LCT * OS is greater than MST - TBO.

$$(0.0001 * 100) > (0.020 - 0.015) \quad (7)$$

Secondly, when the ECR becomes very large, the remote server cannot keep up with the ECR, the receiver will be forced to wait for the server to generate numbers. This can be seen in table 4 for all values where the ECR was 500,000. If the time spent waiting is significantly less than what is required to compute the values locally, then the server decomposition still provides speedup.

## 7. CONCLUSION

The toolkit that we have developed can be used to develop distributed simulation applications without having to invest in new environments or training. SIMSCRIPT and the UNIX operating system are widely available, allowing easy access to these tools. The toolkit is composed of 650 lines of C code and requires about approximately 50 lines of additional SIMSCRIPT code per server/receiver pair to be added to the simulation model. The user of these tools need only be concerned with three C function calls and two SIMSCRIPT routines, so the complexity of the simulation program is not significantly affected.

Use of these tools is most likely to be beneficial in models that can be decomposed into components in which information flow among processors is not cyclic. An example of a simulation model which could benefit from these tools would be a that of a communication network protocol that requires simulated data traffic packets described by a number of time independent complex attributes. Other uses include distributed statistical analysis procedures and graphical displays.

The authors will be glad to provide either printout or e-mail copies of the tookit on request.

## ACKNOWLEDGMENTS

This work was supported in part by CIT under grant INF-89-002-01, by NASA under grant NAG-1-908, and Sun Microsystems under RF596043.

## REFERENCES

USENIX Association (1987), *Unix Programmer's Manual - Supplementary Documents 1.*

USENIX Association (1987), *Unix User Manual.*

Bagrodia, L.R., K.M. Chandy, and J. Misra (1987), "A message-based approach to discrete-event simulation," *IEEE Transactions on Software Engineering*, 13, 6, 654-665.

Cota, B.A. and R.G. Sargent (1986), "Concurrent Programming in Discrete Event Simulation: A Survey," Technical Report, Syracuse University, Syracuse, NY.

Fujimoto, R.M. (1989), "Parallel discrete event simulation," In *Proceedings of the 1989 Winter Simulation Conference*, E.A. MacNair, K.J. Musselman, and P. Heidelberger, Eds. IEEE, Piscataway, NJ, 19-28.

Gimarc, R.L. (1989), "Distributed simulation using hierarchical rollback," In *Proceedings of the 1989 Winter Simulation Conference*, E.A. MacNair, K.J. Musselman, and P. Heidelberger, Eds. IEEE, Piscataway, NJ, 621-629.

Jefferson, D. (1987), "Distributed simulation and the time warp operating system," *ACM SIGOPS.*

Jones, D.W., C.C. Chou, D. Renk, and S.C. Bruell. (1989), "Experience with concurrent simulation," In *Proceedings of the 1989 Winter Simulation Conference*, E.A. MacNair, K.J. Musselman, and P. Heidelberger, Eds. IEEE, Piscataway, NJ, 756-763.

Kaudel, F.J. (1987), "A literature Survey on Distributed Discrete Event Simulation"