Deadlock Avoidance in a Distributed Simulation System

by

Li-Fang L. Hsieh

B.A., National Taiwan University, 1977

A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computing and Information Sciences

KANSAS STATE UNIVERSITY

Manhattan, Kansas

1989

Approved by:

[illegible]

Dr. Virgil E. Wallentine
Major Professor

Table of Contents

Chapter 1

Introduction

1.1. Purposes of the Project

Conventional discrete-event simulation executes sequential computer programs to study the behavior of a physical system. Most of these computer programs are written in special simulation languages, such as Simscript, originally developed by the Rand Corp., or GPSS, IBM's mainframe simulation language [GARZ86].

At the heart of the discrete-event simulation program is an event list, an ordered list containing occurrence times and references to event routines. A control program will select the most imminent event from the event list and pass control to the appropriate event routine. Eventually all the events are scheduled for execution in order of their occurrence times.

The problem with the conventional approach is that it is time-consuming and can not efficiently execute in a parallel processing system. For example, within the last ten years, several parallel processing systems based on large networks of interconnected microcomputers have become commercially available. Multicomputer networks usually consist of hundreds or thousands of nodes communicating with each other in parallel. Using the sequential discrete-event simulation to simulate such large communication networks may require hours or even days of CPU time and not allow multiple events to be scheduled at the same time. In order to achieve better performance, a distributed simulation with no global event list was proposed by Chandy and Misra. Each processor, simulating one node of networks, would execute asynchronously and operate its own simulation clock. The occurrence time of each event would be transmitted to the next processor by a time-stamped message. Theoretically, distribution of the software onto multiple processors, either on loosely-coupled systems or tightly-coupled systems, could make the execution of simulation programs potentially faster and more capable of simulating complex systems.

Since each processor executes one of the distributed simulation programs separately and updates its simulation-time asynchronously, any distributed simulation will have to guarantee that its multiple processors cooperate with each other to ensure that events are eventually properly sequenced. Due to this necessary cooperation restriction, a deadlock problem in which processes are blocked on one another waiting to receive messages could potentially happen in a distributed simulation. Several strategies have been proposed to deal with the problem. In this project, only the deadlock avoidance algorithm proposed by Chandy and Misra [CHAN79] is implemented.

Selecting an appropriate programming language to implement the distributed simulation program, distributed simulator, is very important. Sequential programming languages, either general purpose languages or special simulation languages, are used for conventional single processor simulation. Sequential languages do not provide language-level facilities to create processes on remote processors or to send messages between different processors. In contrast, concurrent programming languages are usually more difficult to debug and to prove correct, but they can express the relationships among parallel processors more naturally, and provide enough language-level facilities needed for distributed simulator. Besides, the processes of a concurrent program can actually be run in parallel if they are run on a multiprocessor system. Even on a single processor, by allowing input/output operations to run in parallel with computation, a concurrent program can reduce program execution time. Using concurrent programming languages to implement a distributed simulator is more appropriate than using sequential programming languages.

In order to explore the performance of distributed simulation, to avoid the problem of deadlock, and to experience the use of a concurrent language, a distributed simulator is implemented. This simulator, written in Concurrent C, uses a deadlock avoidance algorithm, and adopts the basic queueing network scheme which models computer and communication networks. This distributed simulator might elegantly describe networks of multiprocessors in which events actually occur concurrently.

1.2. Contents of the Paper

The rest of this paper is organized as follows. Chapter Two will give a brief overview of the background of the project. The concepts of simulation, physical system, basic queueing network model, distributed simulator, message-passing, and deadlock problems are introduced.

Based on the background introduced in Chapter Two, Chapter Three will present the deadlock avoidance algorithm. A walkthrough of the algorithm is demonstrated. The reason for avoidance of deadlock is explained. The individual algorithm of each process simulating the basic queueing network model is analyzed.

Chapter Four will deal with the implementation details of the project. The available facilities of Concurrent C programming language and the environment of designing the project are introduced. Finally, the whole structure of the distributed simulator is described.

The last chapter, Chapter Five, will give an overall conclusion and several suggestions for possible future work.

Chapter 2

Background

Chapter One highlighted the importance of implementing a distributed simulator for multicomputer networks. This chapter continues a brief discussion of characteristics of distributed simulation. First, the basic concepts of distributed simulation and its three major components: physical system, logical system, and message system [MADI88] are introduced. The basic queueing network model which represents a real system to be simulated in this project is included. Then, the deadlock problem is addressed and a deadlock scenario is examined to illustrate the deadlock that can occur in a distributed simulation.

2.1. Distributed Simulation

Measuring the performance of a system, either to build a new system or to modify an existing one, is difficult. By using a computer simulation of the system, we can study its behavior over a long period of time, get reproducible results quickly, and predict system performance without actually establishing or physically modifying it.

The type of simulation used in computer and communication systems design is discrete-event, system-level simulation. Discrete-event simulation changes the system states at finite points in time, as opposed to continuous simulation which changes.states continuously over time. For example, where the speed of a car is considered, it could be measured over time using a continuous-event simulation, but to know the speed at specific times, discrete-event simulation would be required.

System-level modelling analyzes the system behavior from a register-transfer-level modelling which analyze system behavior from a 'functional' point of view [MACD87]. For example, at the register-transfer-level, the functions of static system components, such as registers, multiplexors, and addresses are evaluated. But at the system-level, the dynamic execution times during which jobs are accomplished are measured in order to study system performance.

A distributed simulation is the execution of discrete-event simulation programs on a parallel processor[REED87]. The structure of the distributed simulation can be divided into three parts: physical system, logical system, and message system. The physical system is the model of the real system to be simulated; the logical system is derived from the physical system; and the message system synchronizes the different clocks in the logical system. Each of them will be discussed respectively in this chapter.

2.1.1 Physical System

2.1.1.1 Modelling

In order to measure the performance of a real system, it must be divided into several distinct functional units. Activities, events, and processes are used in modelling the real system in a discrete-event simulation. The activity is the smallest unit of job in a system. An activity can be triggered and terminated by an event. A group of logically related activities form a physical process. When an activity of a physical process is triggered by an event, the state of the physical process will be updated. The action is called a state transition. Each physical process keeps advancing through states and interacting with others to finish jobs for the system. The execution time for a physical process is the sum of execution times and delay times of all its activities. A whole real system is described at a given level of abstraction by a set of distinct, independent physical processes which group together as a physical system.

2.1.1.2. Basic Queueing Network Model

A basic queueing network model used for the RESQ Simulation Package is adopted as the physical system for this project. In the model, a set of jobs wait in a queue until a server is ready to service them. This model helps to understand the characteristics and effects of congestion in computer and communication systems subjected to both probabilistic and deterministic job flow.

A queue can have one or more servers. A server can have a fixed rate of service time or have a function to calculate a service time based on the state of its queue, such as on the number of jobs at the queue. Usually a service time for a server is given by a probability distribution. There may be one or more sources to create arrival jobs. Each source generates one job at a time. The time between two jobs is called 'inter-arrival' time. It can also be specified by a probability distribution. Apart from the queue, server, and source notations in the network model, there is a sink node which destroys departure jobs, a path arc which shows the flow of jobs, a branch node which distributes jobs among several paths, and a merge node which combines jobs at a certain point in the system. The focus of performance measurement for a basic queueing network model is the amount of time in which jobs have waited in a queue and the amount of time in which jobs are serviced. Figure 2.1 shows the symbols for a basic queueing network model:

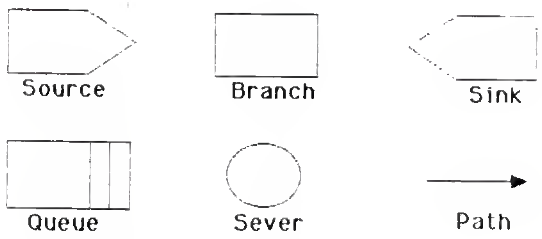

Figure 2.1 Symbols for basic queueing networks

2.1.2. Logical System (Distributed Simulator)

Unlike the sequential, discrete-event simulator consisting of global event routines, the distributed simulator consists of separated logical processes. Each one describes a scenario of actions which represent the behavior of the corresponding physical process. The scenario consists of transformational rules used by the physical process and a description of interactions with other physical processes. The transformational rules of the physical process are modeled by the logic of the logical process. The interactions between physical processes are modeled by time-stamped message-passing between corresponding logical processes. These logical processes of a distributed simulator are executed among parallel processors.

The steps to construct a distributed simulator are as follows:

1.) Program each logical process at a given level of abstraction based on the activities of the corresponding physical process.

2.) Create and execute these logical processes on a multiprocessor computer system.

   The principles of creating and executing processes in a distributed simulator are:

   a.) No central process routes the messages among processors.

   b.) No control process schedules the order of all events of the simulated system.

   c.) No global simulation time controls the speed of all logical processes.

   d.) No global variables are shared among processes.

2.1.3. Message System (Time-Stamped Message-Passing System)

In this project, the computer and communication networks are the real systems to be simulated, the basic queueing network models are the physical systems to model the real systems, and the distributed simulator is the logical system to simulate the physical systems representing the network models. It is assumed that the physical processes modelling multi-computer networks communicate with each other exclusively through messages. To simulate physical processes, logical processes in the distributed simulator can not have shared variables and must communicate exclusively by sending messages. Further, it is assumed that whether each physical process sends out a data as well as the content of the data at an arbitrary time T, depends exclusively on the external messages and internal transformational rules before and at time T. So the logical process sending a message should also depend only on the external messages and the internal logic up to T. The communication rule between two logical processes is implemented as following: A message will be sent from ith logical process to jth logical process, if and only if the ith physical process sends a message to the jth physical process. Along with the message, a simulation time-stamp is sent to synchronize logical processes since the speeds among them are asynchronous.

2.2 Deadlock Problems

In order to achieve a better performance, a distributed simulator will allow logical processes to execute asynchronously (i.e., each logical process updates its local clock without being aware of the speed of other processes). In fact, the actual physical system has to obey a causality principle in order to accomplish jobs.

2.2.1. Casualty Principle

The causality principle states that if a state transition has some effect on another state transition, then the latter must always wait until the former has occurred. In other words, if an event of a process has no direct or indirect cause/effect relationship on another event, either on itself or other processes, then the process can have a state transition. If an event A of a process will cause an event B on itself or other process to be created, then event A has to be processed before event B. This imposes a partial ordering of state transitions in the physical system. To correctly simulate the corresponding physical process, each logical process in the distributed simulator has to obey the local causality constraint, which requires that received jobs are not processed in decreasing time-stamped order. If each logical process does not violate its local causality constraint and the interactions between any pair of logical processes are strictly by message-passing, then the logical processes of the distributed simulator will not violate the causality principles of the physical processes[REED87].

2.2.2. Deadlock

The local causality constraint could cause a deadlock to occur in a distributed simulation. Figure 2.2 illustrates one deadlock situation. In this model, there are one source(A), one branch(B) with two paths(B1 & B2), two queue/servers (C & D), and one sink(E).

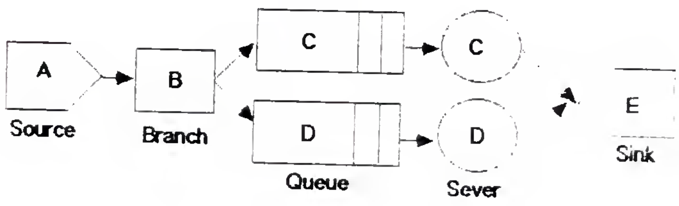

Figure 2.2 Deadlock Scenario

Consider the following scenario:

1.) Source A generates a job (5,m1) and sends it to Branch B.

2.) Branch B selects path B1 to distribute (5,m1) to Queue C.

3.) (5,m1) is stored in Queue C and eventually is serviced by Server C.

4.) Server C sends (15,m1) to Sink E, after adding service-time(10).

5.) Sink E is waiting for a job from Server D.

6.) Source A generates another job (10,m2). The new job is sent through the same path and is stored in Queue C .

7.) A keeps sending jobs to Queue C until C is full.

8.) A wants to send one more job to Queue C.

9.) Branch B gets the job and is waiting for Queue C. At this point, sink E is waiting for server D, server C is waiting for sink E, queue C is waiting for server C, server D is waiting for queue D, queue D is waiting for branch B, branch B is waiting for queue C, and finally source A is waiting for branch B. Deadlock occurs.

2.2.3. Synchronization Algorithms

Several synchronization algorithms, conservative and optimistic [COTA88], have been proposed to address the deadlock problem. Two important conservative algorithms are the deadlock avoidance algorithm and the deadlock detection and recovery algorithm [CHAN81]. The avoidance algorithm avoids deadlock by sending 'NULL' messages to all waiting processes, in order to let these processes extend their local time up to which the state of the simulated physical process is known. The detection and recovery algorithm allows the deadlock to occur, but once it is detected the algorithm generates 'NULL' messages to break the deadlock chain. An optimistic algorithm, the Time Warp algorithm [JEFF85], allows processes to run freely without any constraint. Eventually, some processes would violate their local causalities, at which time a rollback mechanism would discard all erroneous results and restart from the points that violate the constraints. The deadlock avoidance algorithm is chosen for this project.

Chapter 3

Algorithm

Chapter One and Two contained the reasons for implementing a distributed simulator, key issues in designing a distributed simulator, and different strategies to cope with the deadlock problem. This chapter will focus on the deadlock avoidance algorithm, whereas Chapter Four will discuss the implementation in detail. Before discussing the algorithm, a number of assumptions for each logical process in the distributed simulator are made to fit the algorithm and the "lookahead" time used for avoiding deadlock in the algorithm is examined. Then, the algorithm is presented and a walkthrough of the algorithm is given to demonstrate the flow of the algorithm. Finally, based on the algorithm, each individual type of logical process(LP) which simulates the corresponding physical process(PP) of the queueing network model is discussed.

3.1 An Overview

In order to implement the deadlock avoidance algorithm, a number of assumptions are made concerning the behavior of logical processes in the distributed simulation.

A1.) No process can send messages to itself.

A2.) The time-stamp of the first message sent should be greater than 0.

A3.) The last message received should be less than or equal to the prespecified termination time, a simulation time to stop simulating.

A4.) The inter-arrival time should be greater than 0.

A5.) If local time of the process equals 0, then no message should be received; if local time equals the termination time, then a whole stream of messages should be received.

A6.) All messages sent from LPi to LPj at time T are determined by all the messages received by the LPi up to T.

A7.) The propagation delay is assumed to be 0.

A very important component of the algorithm is a function to calculate "lookahead" time for each process at any time T. All output messages sent from LPi to LPj up to time T are dependent on all the incoming messages received up to T by LPi. It is possible to predict all output messages sent from LPi to LPj at time T based upon those messages sent from LPi to LPj at an earlier time T' (T' < T). This implies that all the input messages received by LPi between T' and T will not influence the output messages to LPj before T. (T - T') is the lookahead time for arc(i,j) at time T'. How can we calculate the lookahead time for each process at any time on any arc (i,j)? In other words, how can we compute all the output messages on the arc (i,j) between the current time T' and the time T to which the process can lookahead? According to the deadlock avoidance algorithm, this depends on the state of the physical process i at the current time T' and all the messages received by process i before time T'.

For this project, since the basic queueing model is used to simulate the physical system, it is not possible to run the physical system to get the lookahead time. Instead, a user-specified probability distribution is used to get a service-time for each incoming job, and the service-time is the lookahead-time for the server process.

3.2 General Algorithm for Each LP

Figure 3.1: Logical Process i

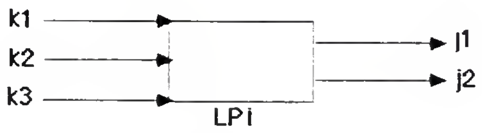

Figure 3.1 shows LPi having three incoming links, k1, k2, and k3, and two outgoing links, j1, and j2. The deadlock avoidance algorithm for each LPi is as follows:

```
1)      {Initialization}:

                For every k DO
```

```
        /*   the time-stamp of the last incoming message from k, the point that in physical
             time up to which the arc(k,i) has been simulated */
             Tki = 0;
        /*   the last incoming message to LPi, no message has been transmitted */
             Mki = NULL;

        For every j DO
        /*   the time-stamp of the last outgoing message to j the point that in physical time
             up to which the arc(i,k) has been simulated */
             Tij = 0;
        /*   the last outgoing message from LPi */
             Mij = NULL;
        /*   the initial clock value for LPi, the time that LPi has simulated its corresponding
             PPi */
             Ti  = Min(Tki,Tij) = 0;
        Z  = Termination Time;

2.)  {Body}:
        Where Ti < Z
        A.)  Selection: Select a set of NEXT input line(s) or/and output line(s) based on the
             local simulation time.
             /*   the smallest time-stamp among all incoming links, the point that the LPi
                  knows a complete input history up to time TIN */
                  TINi  = MIN(Tki);
             /*   the time for LPi to predict all output messages to LPj  */
                  TOUTij  = TINi + Lij(TINi);
             /*   select incoming lines with clock values equal Ti */
                  For every k Do
                  if (Tki = Ti)
                  NEXT = k;
             /*   select outgoing lines with clock values equal Ti and with their predictable
                  output times exceed their last output times */
                  For every j Do
                  if (Tij = Ti) and (TOUTij > Tij)
                  NEXT  = j;

        B.)  Computation : determine the time-stamp and the message transmitted on each
             output line in NEXT.

             For every j in NEXT DO
             /*   the message history for PPi up to time TOUTij, no NULL message */
                  hij(TOUTij) = Fij(TINi,h1i(TINi)...hni(TINi));"
             /*   the message history for PPi, up to time Tij, no NULL message  */
                  hij(Tij)  = Fij(Tij,h1i(Tij)...hni(Tij));
             /*    PPi sent message(s) on line (i,j)*/
                  If hij(Tij) <> hij(TOUTij) then
             /*   Tij<t1<= TOUTij, t1 is the first clock value of message send out by PPi
                  */
                  NewTij  = t1;
                  NewMij  = Message;
```

```
                Else
                      NewTij = TOUTij;
                      NewMij = NULL;
                EndIf
        /*      exceed the termination time */
                If (NewTij > Z) then
                      NewTij = Z;
                      NewMij = NULL;
                EndIf
EndFor
C.)     I/O Operation: Carry out parallel I/O
                operations(by a non deterministic order) for all lines in NEXT.
        /*      Do 1 & 2 in parallel */
                1.)   For every j in NEXT
                      /*    Wait to send messages out */
                            Tij = NewTij
                            Mij = NewMij
                2.)   For each input k in NEXT
                      /*    Wait for input */
                            Tki = NewTki
                            Mki = NewMki
D.)     Compute Ti:
        Ti = MIN(Tki,Tij);
```

3.3 Walkthrough of Algorithm

Assume for the Figure 3.1 above that the sequence of messages for each incoming link, k1, k2, and k3 are as below. The format of each message is composed of time-stamp, message, and message destination. The lookahead time(or service time) for LPi is fixed, equal 5 time units. Table 1 shows the times when each message crosses the corresponding link of the physical system. TINi, TOUTi and Next are defined in the algorithm. Qj12 is the queue to store received messages which can not be sent out at the moment. This is because up to that time the simulated PPi has not sent these messages along its outgoing links yet.

k1 :	(1,M1,j1)	(4,M2,j2)	(10,M3,j2).......
k2 :	(3,M4,j2)	(8,M5,j2)	(18,M6,j2).......
k3 :	(5,M7,j1)	(6,M8,j1)	(15,M9,j1).......

TABLE I. Walkthrough of Algorithm

Step	TINi	TOUTi	NEXT	k1	k2	k3	j1	j2	I	Qj12
1				0	0	0	0	0	0	0
2	0	0	k1, k2, k3,	1	3	5	0	0	0	0
3	1	6	j1, j2,	1	3	5	6,R 6,N	1	0	
4	1	6	k1	4	3	5	6	6	3	0
5	3	8	k2	4	8	5	6	6	4	8
6	4	9	k1	10	8	5	6	6	5	8,9
7	5	10	k3	10	8	6	6	6	6	8,9,10
8	6	11	k3, j1, j2,	10	8	15	8,N 8,R	8	9,10,11	
9	8	13	k2, j1, j2,	10	18	15	9,N 9,R	9	10,11,13	
10	9	14	j1, j2	10	18	15	10,R 10,N	10	11,13	

Step 1:

Initially, both ks and js are set to 0 which indicates that the lines have been simulated up to time 0.

Step 2:

TINi is 0. There is no lookahead time. k1, k2, and k3 are selected for accepting messages with time-stamp 1, 3, and 5 respectively.

Step 3:

TINi is 1, which indicates that the message with time-stamp 1 is selected to be sent. After adding the service-time 5, TOUTi becomes 6. Both j1 and j2 are chosen to send out messages with the same time-stamp 6, but ji gets a Real message and j2 gets a Null one. Update the local time for LPi to 1.

Step 4, 5, 6 and 7:

Messages with time-stamp 4, 8, 10, and 6 are received. Message with time-stamp 3, 4, and 5 are queued with their new output times 5, 8, and 10 respectively. The local time is advanced each step.

Step 8, 9, 10.:

Several new messages are received. The message sent out each step is the first message stored in the queue.

3.4 Avoid Deadlock

Let us consider the scenario of Section 2.2.2 in Chapter Two. Let us see how this algorithm avoids deadlock in that scenario:

1.) Source A generates a job (5,m1) and sends it to Branch B.

2.) Branch B selects path B1 to send (5,m1) to Queue C. Meanwhile Branch will send (5,NULL) to Queue D.

3.) (5,m1) is serviced by Server C. (5,NULL) is serviced by Server D.

4.) After adding service_time(10), Server C sends (15,m1) to Sink E. After adding service_time(5), Server D sends (10,NULL) to sink E.

5.) Sink E selects (10,NULL) and processes it. The deadlock situation showed in chapter 2 will thus be avoided by sending a NULL message to the unselected path.

3.5 Structure for Each Type of LP

Five types of logical processes--source, branch, queue, server and sink-- are implemented in this distributed simulator. Each one corresponds to the physical process in the queueing network model.

3.5.1. Source

A source process either generates a new job or sends it out into the system being modeled. Figure 3.2 shows the algorithm of the source process. The inter-arrival time between two jobs is determined by a user-specified probability distribution. According to the algorithm, the source process will guarantee that the inter-arrival time is greater than 0. The time-stamp for each new job will be its inter-arrival time plus the current local time. If the time-stamp of this new job exceeds the specified termination time, then a NULL message with the termination time will be sent out. The source process keeps track of the specified interval time for sending out a statistical report. The Concurrent C code for the source process (Source.cc) is shown in the Appendix A.

Figure 3.2 Source algorithm:

```
Process Source( )
   begin
    Accept Setup( );
     Time = 0.0
     Generate_time = 0.0
     Out_time = 0.0
    Loop
     If (Generate_time = Time)
        Msg_time = Time + Inter_arrival
        Msg_id = Msg_id++
           Msg_type = REAL
           Generate_time = Msg_time
     ElseIf (Out_time = Time &&
              Generate_time > Out_time)
        If (Msg_time > Term_time)
             Msg_time = term_time
          Meg_id = -1
          Meg_type = NULL
        EndIf
        Send(Msg)
           Out_time = Msg_time
     EndIf
     Time = MIN(Generate_time, Out_time)
     If (Statistical_Interval)
       Send(Stats)
    End Loop
 End Process Source
```

3.5.2. Branch

A branch process receives a job and selects one out of all its outgoing links to send the job. Figure 3.3 is the algorithm of the branch process. The maximum incoming or outgoing links are limited to 5. The selection of an outgoing link is based on the user's specified link probability instead of the comparison of the time-stamps. According to the deadlock algorithm, each time the branch process sends a real message out along one of its out links, it has to send a NULL message with the same time-stamp among all the unchosen links. The branch process does not collect any statistical report in the simulator. The Concurrent C code for the branch process(Branch.cc) is shown in Appendix A.

Figure 3.3 Branch algorithm:

```
Process Branch( )
   begin
    Accept Setup( );
     Time = 0.0
     In_time = 0.0 (for each IN link)
     Out_time = 0.0 (for each OUT link)
    Loop
       Outmsg = Smallest incoming time-stamped Msg
      For each IN link
        If (In_time = Time)
          Accept(Msg)
            In_time = Msg_time
      For each OUT link
        If (Out_time = Time && Outmsg_time > Out_time)
          If (Selected)
            Send(Outmsg_time, Outmsg_id)
          Else
            Send(Outmsg_time,NULL)
          Out_time = Msg_time
     Time = MIN(In_time, Out_time)
    End of Loop
  End of Branch Process
```

3.5.3. Queue

The queue process accepts a job and stores it into its queue until its associated server requests the job. Figure 3.4 shows the algorithm of the queue process. In the distributed simulator, only one server is associated with each queue and the queue size is not infinite. Basically, the functions of each queue can be divided into three:

1.) As long as the queue is not full, the queue process can select the smallest time-stamped job(s) and put it into its queue. If the numbers of the smallest time-stamped jobs are greater than the available spaces in the queue, the extra jobs will be stored in a temporary queue. Once a space is available in the queue, jobs stored in the temporary queue will be moved immediately in a FIFO order.

2.) As long as the queue is not empty, the queue process is ready to accept a "job request" from its server. Based on a user-specified method, such as FIFO, the queue process sends out the desired job and adjusts its queue to be ready for accepting a next request. The current local time will be the time-stamp of the job sent out.

3.) Whenever its server requests a statistical report, the queue process will calculate the necessary information and give it to the server.

According to the deadlock algorithm, an assumption was made for the protocol between two logical processes: a message is sent from logical Process i to logical Process j if and only if Process i is ready to send and Process j is ready to receive. In fact, the function of the queue process is quite passive. It accepts and sends jobs with several conditions. For input, the queue must not be full. For output, the queue must not be empty (Process i is ready), and a request is received from its server(Process j is ready). If the queue is full, even though the incoming link equals its local time, it cannot accept jobs. If the queue is empty, even if a request is received, the queue process has no job to send. First, the queue process can not do I/O operations based only on the comparison of the link times and its local time. Second, local time is not the minimum value between incoming links and outgoing links. Instead, it is the minimum value between incoming link time and the time-stamp of the last message stored in the queue. Third, local time will not be influenced by the time at which a server sends a job request. The Concurrent C code (Queue.cc) for the queue process is shown in Appendix A.

Figure 3.4 Queue algorithm:

```
Process Queue( )
   begin
    Accept Setup( );
     Time = 0.0
     In_time = 0.0 (for each IN link)
     Stored_time = 0.0
    loop
       StoredMsg = Smallest time-stamped incoming message

      If (queue is not Full)
        For each IN link
          If (In_time = Time)
            Accept(Msg)
                In_time = Msg_time
          If (Stored_time = Time && In_time > Stored_time)
                Store(Msg)
                Stored_time = Msg_time
            If (#Msg > #Available_Space)
                  Temporary(Msg)
          Time = MIN(In_time,Stored_time)
        EndFor
      EndIf
      If (queue is not empty)
           Accept(Request a Msg)
           Select(Msg)
           Msg_time = Time;
           Send(Msg)
      If (Request a stats report)
           Accept(request)
           Send(Stats)
  End of Loop
End of Queue Process
```

3.5.4. Server

The activities of a job are typically focused on servers. A server process requests a job from its own queue, generates a service time based on a user specified probability distribution, adds the service time onto the original time-stamp of the job and sends it out. Figure 3.5 is the algorithm of the server process. In this distributed simulator, the service time is the lookahead time for the server process. Based on the assumption of the algorithm, increasing TINi will not decrease TOUTi. The source process in this simulator will guarantee a reasonable service time to each job. The algorithm of the server process is implemented exactly the same way as the deadlock avoidance algorithm described above.

When the interval time for a statistical report expires, the server process requests information from its queue process, incorporates its own, and sends it out. The Concurrent C code(Server.cc) for the server process is shown in Appendix A.

Figure 3.5 Server algorithm:

```
Process Server( )
    begin
     Accept Setup( );
      Time = 0.0
      In_time = 0.0
      Out_time = 0.0
     loop
          Tin = In_time;
          Predict = Tin + Service_time
       if (In_time = Time)
             Accept(Msg)
             In_time = Msg_time
       If (Out_time = Time && Predict > Out_time)
             If (Queue_Msg)
               Msg_time = Queue_Msg[1]_time
             Else
               Msg_time = Predict
             Send(Msg)
             Out_time = Msg_time
       Else if (New_msg && Out_time != Time)
             Queue(Msg)
       EndIf
       Time = MIN(In_time, Out_time)
       If (Stats_Interval)
            Send(Stats)
  End of Loop
End of Server Process
```

3.5.5. Sink

The sink process destroys one job at a time from the modeled system. Figure 3.6 is the algorithm for the sink process. If the job is not a NULL job then the numbers of sunk job will be increased. Only one sink is necessary to provide departures of jobs in our simulator. The Concurrent C code(Sink.cc) for the sink process is shown in Appendix A.

Figure 3.6 Sink Algorithm:

```
Process Sink( )
   begin
    Accept Setup( );
     Time = 0.0
     In_time = 0.0
     Out_time = 0.0
    loop
      if (In_time = Time)
            Accept(Msg)
            In_time = Msg_time
      If (Out_time = Time && In_time > Out_time)
            Sunk(Msg)
            If (Msg_type <> NULL)
              Num_sunk + I
            Out_time = Msg_time
      Time = MIN(In_time, Out_time)
      If (Stats_Interval)
           Send(Stats)
  End of Loop
End of Sink Process
```

Chapter 4

Implementation

The previous chapters discussed characteristics of the distributed simulator and the algorithm used to solve the deadlock problem. The algorithm of logical processes simulating physical processes is only the foundation of a distributed simulator; without a programming language to implement the algorithm, it only represents a blueprint. Even after the logical processes are implemented by a programming language, without several supplemental processes to support them, they can only build an engine, not a complete simulator. For instance, to measure the performance of a physical system, the distributed simulator must have information representing the characteristics of the physical system, and information representing the behavior of that system. This chapter first introduces important language-level facilities provided by Concurrent C, then discusses the original design environment. Finally, it introduces three different functions of the supporting processes of the distributed simulator, creation of processes, collection of reports, and termination of processes.

4.1 Programming Language (Concurrent C)

Concurrent C was selected as the programming language to implement this project, because it has more application-level facilities needed in a distributed simulator than other concurrent programming languages and is compatible with the available multiprocessor system.

The fundamental building blocks of the Concurrent C language are processes. Each process consists of two parts: a type (specification) and a body (implementation). The process type declares all information necessary to create and interact with other process types. The process body is executed by that process type. For example, a source process body is executed by each instance of the source process type.

In this project, only the distributed aspect of the simulation is of interest. The rest of the characteristics of the simulated physical system are encapsulated. Five types of logical processes--Source, Branch, Queue, Server, and Sink--which simulate the physical processes in the queueing network model are the major components of the distributed simulator. The remainder are supporting processes used to create processes, collect statistical information, and terminate the simulator.

Processes can be dynamically created in a distributed system. To create processes on remote processors, Concurrent C uses a built-in function declared as:

```
processor_pid = c_processor(char *machine, char *program);
```

This function creates a new logical processor and returns its processor id. The parameter 'machine' is the processor name and the parameter 'program' is the path name of a load module on this processor[GEHA88]. Placing the returned processor_id into the following expression would create one instance of a source process on a particular remote processor:

```
source_pid = create Source( ) processor(processor_pid)
```

The interactions between two processes are done totally by transaction calls in Concurrent C. A process that initiates a call is called a caller. A process that receives the call is called a receiver. There are two kinds of transaction calls, asynchronous and synchronous, demonstrated respectively:

```
async trans setup( );

trans struct Msg_Rec  get_Msg( );
```

For an asynchronous call, the caller can resume its execution immediately after submitting a transaction call. For a synchronous call, after a caller submits a call, it must wait until the receiver accepts the call, performs actions, and sends back data. Then the caller can continue to execute its code.

The parallel I/O operations in the deadlock avoidance algorithm cannot be implemented using Concurrent C transaction calls, because a process cannot be accepting a call while submitting or accepting another call, or vice versa. A process can only accept or submit one call at a time. But an asynchronous transaction call can be used to limit the message-passing delay, if a caller does not need a return value from its receiver. For instance, 'setup' calls are used only to send original parameters to each logical process. No return value is needed. But 'get_Msg' calls are used by a server to get a job from its queue. Before getting a job, the server cannot resume it execution.

The statement for accepting a call at the receiver site has the form:

```
accept send_msg( ) suchthat( ) by( )
```

The 'suchthat' and 'by' clauses are two alternatives for selecting a call among all pending transaction calls. The 'suchthat' clause is a conditional expression to the first pending transaction call which satisfies that condition. If none of the calls satisfies the condition, then all calls are held until an appropriate call arrives. The 'by' clause is a priority expression. All pending transaction calls will be evaluated by the expression to decide which one has the lowest value. The other calls are held to be executed at some later time. Both expressions play very importment roles in implementing the deadlock avoidance algorithm. The accept 'send_msg' statement in the following example is used for each logical process to select a transaction call. In figure 4.1, by using 'suchthat', one can actually select the acceptance of the next message on a desired incoming link. When the desired incoming link has not been assigned to any of its specified caller, one of first messages sent from any caller will be accepted. Otherwise, the caller of the message must be the same with the assigned caller of the desired link. After selecting the messages on a desired link, 'by' is used to select the lowest time-stamped message on that chosen link.

```
for(;;) {
  select {
    (!Done):
      accept send_msg(Job)
            suchthat((In_Line[n].Caller == NONE &&
                 Job.id == 0) ||
                   Job.from == In_Line[n].Caller)
            by (Job.send_time)

  or   accept term( )
              terminate;
  }
}
```

Figure 4.1 Code for Receiving Incoming Messages

The 'select' statement shown above can be used for logical processes to wait for the first arrival call among several different types of transaction calls. The (!Done) guard must be true before a send_msg call is accepted. Only one alternative will be chosen at a time. Processes can be dynamically terminated in a distributed way too. To terminate a process on a remote processor, Concurrent C uses a transaction call to tell the process to be ready to terminate. If a process completes its code or the 'terminate' is one of the alternatives inside of the 'select' statement, then it can take the termination transaction call and break the for-loop. Until all processes terminate, then the whole simulator terminate collectively. The example above shows that if any process selects to accept a transaction call from the Terminate process, then it is ready to terminate.

For this distributed simulator, all processes are implemented in Concurrent C processes. All logical processes representing physical processes in the basic queueing network model can be created and terminated in a distributed way. All communications between processes are done by transaction calls.

4.2 Environment of Project

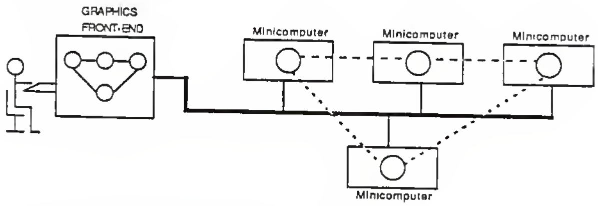

Figure 4.2 Distributed Simulation Environment[VOPA89]

This distributed simulator is designed for an environment, shown in Figure 4.1, consisting of a Xerox workstation, Sun workstations, Vax(DEX 11/780), Harris(HCX-9), and a set of minicomputers (3b2/3b15's). Users can build up a visual basic queueing network model on the Xerox graphical front-end by using a set of icons and specifying some desired information, such as the probability distribution for Source or Server processes. After the job is done, an Internet socket is connected from the Xerox to one of the minicomputers where the main program of the distributed simulator is executing. Once the connection is established, the input information is sent from the

Xerox. This simulation project begins from the reception of that input information. The simulator creates logical processes and distributes them among several machines according to the user's specification, and runs the simulator parallelly under the deadlock avoidance algorithm. When a prespecified time for a statistical report comes, all the needed information will be collected and sent back to the Xerox front-end.

The following, Figure 4.3, is the flow for this distributed simulator:

```
          Distributed Simulator

    1.) Create a socket and wait for input (Connect.c)
    2.) Read input (InOut.cc)
    3.) Parse and store the initial information (Build.cc)
    4.) Create all processes (Create.cc)
    Loop
       Parallel Operations(5 and 6)
        5.) Run all processes parallelly (Source.cc,
             Branch.cc, Queue.cc, Server.cc, Sink.cc,
              Collector.cc, and Terminate.cc)
        6.) Collect a statistical report and send
             back to the user. (Collector.cc and
              InOut.cc)
              Wait for control message from user
               (Collect.cc)
                  If "terminate simulation" then
                   call termination (Terminate.cc)
                  else "continue simulation" then
                   keep looping
    Until (termination time) or
         (Allowed jobs have been generated)
```

Figure 4.3 Flow of the Distributed Simulator

4.3 Supporting Processes

4.3.1 Creation of processes

The way of handling process creation in this simulator has been influenced by the work of Edward Vopata's thesis, "Distributed Discrete-Event Simulation in Concurrent C" [VOPA89].

The distributed simulator starts with opening a socket to accept an input file. If the input comes from the Xerox front-end, an internet socket is connected. Otherwise, a standard in/out socket is used. The input format is shown in Vopata's thesis[VOPA89]. Once the connection is established, input information is read in, one line at a time. Each line is then parsed one token at a time. Tokens parsed from the same line represent the characteristics of one of the physical process. Therefore, the whole system is represented by the collection of all tokens. According to the specification of a simulated physical system, the simulator creates all necessary processes in a distributed or centralized manner.

4.3.2. Collection of Reports

The Collector process collects a complete statistical report at each specified time interval and sends it to the user. Each process, except the Branch process, sends a statistical report at each specified time interval to the Collector process based on its local time. Because the local time of each process is not updated by a fixed constant value, each report can not be sent exactly at each time interval. Division is used to determine whether a logical process should send a report at the current time. The initial iteration value for time interval equals 0. Each time the value changes, a statistical report is sent. For example, the time interval is assumed to be 15. The local time of source process X is currently increased from 14 to 22. Because 22 / 15 = 1, the first report is sent from X. Let us assume that the local time of another source process Y is increased from 10 to 20. A report is also sent from Y. Two statistical reports are being sent at the same time interval, in fact, the two local times of two sending processes might be slightly different. Sometimes, the time between two reports sent from a logical process might not exactly equal the specified time interval. From previous example, the last report was sent by source process X at its local time 22. Now at a local time assumed to be updated to 32, a new report is sent. In this case, the time between two reports is 10.

Suppose, the current local time of source process X is updated from 32 to 60, 60 / 15 = 4, a report with iteration value 4 is sent. The report with iteration value 3 would be skipped. In this case, because the report with iteration value 3 has not been received completely, the

Collect process cannot send it to the user. In order not to skip too many reports, in this distributed simulator, once the Collector process recognizes that one or more reports are skipped from a logical process, the previous report sent by that process would be used to fill in the skipped iteration.

The Collector process receives each individual report from each logical process, checks the iteration value of the time interval of the received report, and collects all the reports have the same iteration value together. Once the report is completed, it is sent. The format of the statistical report is shown in Vopata's thesis[VOPA88].

4.3.3. Termination of Processes

The Terminate process submits transaction calls to all processes to be ready to terminate. The following two conditions would cause the Collector process to activate the Terminate process.

1.) After sending a statistical report to the user, if a "termination simulation" control message results, the Collector process would initiate the Terminate process. It would signal Source processes first. Source processes would then generate a distinct type of messages, called "Term_Msg" with a negative time-stamp, and send them to the rest of the logical processes. Once a process receives a "Term_Msg", it is ready to accept a transaction call from the Terminate process. Gradually, all processes would be terminated.

2.) If one of the processes passes the termination time and is ready to terminate, the process uses a final statistical report to notify the Collector process. When the Collector recognizes that final statistical reports have been sent by all the necessary processes, it initiates the Terminate process. At the time, the local times of all logical process should be the specified termination time and all processes should be at a ready-to-be terminated stage. No "Term_Msg" would be sent by the Source process in this case.

A source process has two different schemes to determine when to stop generating new jobs.

1.) The termination time is specified to 0, meaning that the simulation time is infinite. (In this case, the number of jobs allowed to be generated by each source process must not be 0.) When the amount of allowed jobs have been generated by the source process, the Source process stops generating new jobs.

2.) The number of jobs allowed to be generated is equal to 0, meaning that the source can generate infinite jobs. (In this case, the termination time must not be specified to 0) The source process stops generating new jobs only when its local time equals the specified termination time.

Chapter 5

Conclusion and Future Work

The design and implementation of the distributed simulator have been discussed in the previous chapters. This last chapter will conclude with several suggestions for future works. In addition, the problem of the C compiler on the 3b2/3b15's is explained.

5.1 Problem

In order to distribute processes of the Concurrent C distributed simulator, all machines must be compatible. Therefore, the distribution of the simulator is designed on 3b2/3b15's. The Concurrent C compiler generates C code and then let the C compiler finish the final compilation. The C compiler on the 3b2/3b15's, AT&T version 4.1, limits the sizes of parameters and messages to 4 K-bytes. Any attempt to pass a parameter or message with size larger than 4 K-bytes will cause a run-time error message 'core dump' to occur. The bug was reported and certified by AT&T Software Support and should be corrected in version 4.3 of the C compiler [VOPA88]. Because the C compiler on the 'ksuvax1', DEC Vax 11/780, is version 4.3, the speed is faster than 3b2/3b15's, and the debugger, DBX, is more efficient than the SDB debugger on the 3b2/3b15's, eventually the distributed simulator is run on the ksuvax1. One example of an input model and its final statistical result is shown in the Appendix B.

5.2 Conclusion

The simulator written in Concurrent C is a distributed version, in contrast to other sequential language simulators. The speed of the Concurrent C simulator is supposed to be faster than the sequential one, but so far, the Concurrent C distributed simulator does not dramatically improve its speed.

The first key to improving speed is to minimize the amount of communication required among distributed processes. If the inter-machine communication is kept to a minimum, distributed processing could be more efficient. But in the case of the distributed simulator, the main activity of the processes is sending messages. This means there will be a large amount of inter-machine communication. Sending a message to another machine is very expensive in terms of time, but that is the only way the simulator communicates. The best approach is to try to partition the simulation model, such that inter-machine communication is kept to a minimum.

The second key to improving speed is to run the Concurrent C distributed simulator on proper machines to reduce the amount of message passing. For example, the new version of 3b15, Apache, is a tightly coupled system and contains a hypercube structure which can hold many processes. The 3b2 machine, a loosely coupled system, used for this project, cannot handle more than one unix process. But if a heavily computational intensive program running on one machine, would demonstrate the difference.

However, a distributed simulator is just one application of distributed programming. For this project, we are interested in distributed Concurrent C in terms of writing distributed programs as opposed to testing the efficiency of distributed programs.

5.3 Future Work

A basic queueing network is not in itself sufficient to represent a complex system. Several facilities[SAUE80] can be added on to the distributed simulator. For example,

A.) Add job variables used to store data associated with each individual job. Two types of job variables, class and phase, can be tagged to each job. The class variable defines to which class the job belongs. At branch nodes, the job's class variable can be used to determine the routing, then a lot of redundant nodes in our graphics can be reduced. The phase variable identifies a job's current position when the job flows between the servers within the model. By the existing id variables and the new phase variables, more statistical output can be obtained and more about the flow situation can be understood.

B.) Add Resource nodes(a pool of tokens) which represent a passive queue. There are several related nodes that can be added to Resource nodes. An Allocate node allows jobs to request possession of a number of the tokens from particular resources. A Release node allows jobs to return all tokens which it holds back to particular resources. For example, a Resource node can be added in the simulated system to hold a pool of buffer tokens, with an Allocate node in front of the server and a Release node behind the server. Before requesting a service, a job must request for buffer tokens. After leaving the server, that job will release the tokens back to the Resource node.

C.) Add Set nodes defined to have one or more assignments for job variables in the programming language sense. A job visiting this node causes the assignment statements associated with that Set node to be performed. For example, a Set node can be added in front of a Server to let jobs of one of the classes request a modification to its service time. After getting the disired service, the job might pass through the Set node first, and request a new service time by giving a new distribution function. Currently, the distribution function for the service time in this queueing model is fixed.

D.) Add Decide nodes on Branch nodes which decide which route the job is going to send by the job variable, not by probability.

E.) Add User Nodes(Code Segments) which can invoke a user-written C function. A heavily computational function can be put in this node, making it easier to verify whether this Concurrent C distributed simulator is faster than simulators written in other languages.

F.) Add other features:

- 1.) Ending simulation (or printout statistical information)
    - 1.) after X jobs are serviced by a server
    - 2.) after Y jobs are sunk by a sink
- 2.) Printing statistical information only for specific nodes, not all notes.
- 3.) During a simulation run, it is possible to interact with the model by changing parameters and then resuming the run.

G.) Compare the performance of distributed simulators implemented using three different synchronization algorithms, two conservative algorithms (deadlock avoidance, and deadlock detection and recovery) and one optimistic algorithm (Time Warp).

Bibiography

[CHAN79] K. M. Chandy & J. Misra, "Distributed Simulation: A Case Study in Design and Verification of Distributed Programs," IEEE Transactions on Software Engineering, Vol. SE-5, No. 5, September 1979, pp. 440-452.

[CHAN81] K. M. Chandy & J. Misra, "Asynchronous Distributed Simulation via a Sequence of Parallel Computations," Communications of the ACM, Vol 24, No. 11, April 1981, pp. 198-206.

[COTA88] B. A. Cota, & R. G. Sargent, "An Algorithm for Parallel Distributed Event Simulation Using Common Memory," Annual Simulation Symposium, pp. 23-31.

[GARZ86] R. F. Garzia, & M. R. Garzia, "Discrete-event Simulation," IEEE Spectrum ,December 1986, pp. 32-36.

[GEHA88] N. H. Gehani, & W. D. Roome, Concurrent C, AT&T Bell Laboratories, (Submitted for Publication).

[JEFF85] D. R. Jefferson, "Virtual Time," ACM Transactions on Programming Languages and Systems, Vol. 7, No. 3, July 1986, pp. 404-425.

[MACD87] M. H. MacDougall, "Simulating Computer Systems: Techniques and Tools" The MIT Press , 1987. pp. 1-28.

[MADI88] V. Madisetti, J. Walrand, & D. Messerschmitt, "Efficient Distributed Simulation," Annual Simulation Symposium , pp. 5-21.

[REED87] D. A. Reed, & R. M. Fujimoto, " Multicomputer Networks: Message-Based Parallel Processing," The MIT Press , 1987. pp. 239-267.

[SAUE80] C. H. Sauer, & E. A. MacNair, "Simulation of Computer Communication Systems," Prentice-Hall, INC., , pp. 9-14.

[VOPA89] E. Vopata, "Distributed Distributed-Event Simulation in Concurrent C" Masters Thesis, Kansas State University , 1989.

Appendix A: Concurrent C Source Code for the Distributed Simulator

```
# Makefile
CCC=/usrb/scott/ccc/bin/CCC
CCCLIB=/usrb/scott/ccc/lib/libmpcc50g.a

# add source file names here
SRCS = Main.cc Inout.cc Build.cc Create.cc Distrib.cc Stats.cc Source.cc
       Branch.cc Queue.cc Server.cc Sink.cc Collector.cc
       Terminate.cc

# add same name but with ..o on the end here

OBJS = Main..o Inout..o Build..o Create..o Distrib..o Stats..o Source..o.br
       Branch..o Queue..o Server..o Sink..o Collector..o
       Terminate..o

HDRS = define.h distrib.h lp_info.h lp_param.h lp_stats.h
       mach.h msg.h rand.h spec.h

# flags for CCC here (-g does symbol generation for dbx)
CFLAGS = -g
# CFLAGS = -g -DSYS5

# Libraries
LIBS = -lm
# LIBS = -lnet -lm

a.out: ${OBJS} ${HDRS}
       CCC ${CFLAGS} ${OBJS} -o a.out $(LIBS)

${OBJS}:
       CCC ${CFLAGS} -c $*cc

# say "make print" will create a file called "print" with files pr-ed together
print: ${HDRS} ${SRCS}
       pr ${HDRS} ${SRCS} > print

# say "make depend" to automatically create dependencies at bottom of this file depend:
       cat </dev/null >.x.c
       for i in ${SRCS}; do
      (echo 'basename $$i .cc'..o: $$i >>makedep;
            /bin/grep '^#[  ]*include' .x.c $$i | sed
                -e 's,<)>,"/usr/include/1",'
                    -e 's/:[^"]*""]*)".*/: 1/'
                -e 's/.cc/..o/' >>makedep); done
    echo '/^# DO NOT DELETE THIS LINE/+2,$$d' >eddep
     echo '$$r makedep' >>eddep
     echo 'w' >>eddep
     cp Makefile .Makefile.bak
     ed - Makefile < eddep
     rm eddep makedep .x.c
     echo '# DEPENDENCIES MUST END AT END OF FILE' >> Makefile
     echo '# IF YOU PUT STUFF HERE IT WILL GO AWAY' >> Makefile
     echo '# see make depend above' >> Makefile

# DO NOT DELETE THIS LINE -- make depend uses it

Main..o: Main.cc
Main..o: define.h
Inout..o: Inout.cc
Build..o: Build.cc
Build..o: define.h
Create..o: Create.cc
Create..o: /usr/include/netdb.h
Distrib..o: Distrib.cc
Distrib..o: /usr/include/math.h
Distrib..o: rand.h
Source..o: Source.cc
Branch..o: Branch.cc
Queue..o: Queue.cc
Server..o: Server.cc
Sink..o: Sink.cc
Collector..o: Collector.cc
Collector..o: /usr/include/stdio.h
Terminate..o: Terminate.cc
# DEPENDENCIES MUST END AT END OF FILE
# IF YOU PUT STUFF HERE IT WILL GO AWAY
# see make depend above
```

```
/    ****************************************************************    */
/*   Include File define.h                                               */
/    ****************************************************************    */
/*                                                                       */
/*   This include file declares constants for process types,            */
/*   queue methods, user commands, message types, report                */
/*   types, number of links and length per line, signals for            */
/*   termination, and all other includes files needed for the           */
/*   simulation.                                                         */
/*   This file is included in each file of the simulator.               */
/*                                                                       */
/    ****************************************************************    */


#define   TRUE            1
#define   FALSE           0

#define   SOURCE                   0   /*  Source process type          */
#define   SINK                     1   /*  Sink process type            */
#define   QUE_SRV                  2   /*  Q/Server process type        */
#define   BRANCH                   3   /*  Branch process type          */

#define   FIFO                     0   /*  First in first out          */
#define   LIFO                     1   /*  Last in first out           */
#define   SIRO                     2   /*  Service in random order      */
#define   PRIO                     3   /*  Highest priority             */
#define   PROB                     4   /*  Probability                  */

#define   TERM                    96   /*  Terminate simulation         */
#define   CONT                    97   /*  Continue simulation          */
#dcfine   START                   98   /*  Start of input file          */
#define   BEGIN                   99   /*  Begin of LP's parameters     */
#define   END                     99   /*  End of LP's parameters       */

#define   NULL_MSG                 0   /*  Null message                 */
#define   REAL_MSG                 1   /*  Real message                 */
#define   TERM_MSG                 2   /*  Termination message          */

#dcfine   STATS_NORMAL             0   /*  Normal stats report          */
#define   STATS_FINAL              1   /*  Final stats report           */

#define   MAXLINK                  5   /*  Maximum I/O links            */
#define   MAXSIZE                100   /*  Maximum queue size           */
#define   MAXLINE                 70   /*  Maximum length per line      */
#define   MAXMACH                 16   /*  Maximum number of machine    */
#define   MAXLP                  110   /*  Maximum number of LP         */
#define   COL_ID             (MAXLP)   /*  Collector process id         */
#define   TERM_ID          (MAXLP+1)   /*  Terminate process id         */
```

```
#define    TERM_TIME                  -0.1   /*    Signal for termination          */
#define    TERM_STAMP                 -0.2   /*    Signal for termination          */


#define    MIN(a,b)
#define    MAX(a,b)


#include <stdio.h>
#include <math.h>

#include "distrib.h"
#include "msg.h"
#include "lp_info.h"
#include "lp_param.h"
#include "lp_stats.h"
#include "spec.h"
#include "rand.h"

  /* From Vopata's thesis[VOPA89] */
#ifdef SYS5  /* For 3b2's and 3b15 Machines (System V systems)


                           char *strchr();    /*    == BSD's index        */
                           char *strrchr();   /*    == BSD's rindex       */
#     define index(a,b)    strchr(a,b)        /*    Map index to strchr   */
#     define rindex(a,b)   strrchr(a,b)       /*    Map rindex to strrchr */

#else  /* For BSD systems */

char    *index();    /* BSD index  : find char forward search    */
char    *rindex();   /* BSD rindex : find char reverse search    */

#endif


char *malloc( );
long  time( );
```

```
/*     ****************************************************************     */
/*     Include File distrib.h                                               */
/*     ****************************************************************     */
/*                                                                          */
/*     This include file declares constants for each probability            */
/*     distribution and its parameters.                                     */
/*                                                                          */
/*     ****************************************************************     */


#define    FIXED          0
#define    UNIFORM        1
#define    POISSON        2
#define    BINOMIAL       3
#define    EXPNTL         4
#define    NORMAL         5
#define    GAMMA          6
#define    BETA           7
#define    ERLANG         8
#define    LOGNORMAL      9
#define    WEIBULL        10

struct Fixed_Rec   {
      double   time;
};

struct Uniform_Rec {
      long   lower;
      long   upper;
};

struct Poisson_Rec {
      double   mean;
};

struct Binomal_Rec {
      long    num;
      double   prob;
};

struct Expntl_Rec {
      double   mean;
};

struct Normal_Rec {
      double   mean;
      double   stdev;
};
```

```
struct Gamma_Rec {
      double  mean;
      double  k;
};

struct Beta_Rec {
      double  k1;
      double  k2;
};

struct Erlang_Rec  {
      double mean;
      long    k;
};

struct Lognormal_Rec {
      double mean;
      double stdev;
};

struct Weibull_Rec {
      double shape;
      double scale;
};

struct Dis_Rec {

      int          Dis_Type;
      double       Min;
      double       Max;


        union  Dis_Data {
          struct Fixed_Rec      Fixed;
          struct Uniform_Rec    Uniform;
          struct Poisson_Rec    Poisson;
          struct Binomal_Rec    Binomial;
          struct Expntl_Rec     Expntl;
          struct Normal_Rec     Normal;
          struct Gamma_Rec      Gamma;
          struct Beta_Rec       Beta;
          struct Erlang_Rec     Erlang;
          struct Lognormal_Rec  Lognormal;
          struct Weibull_Rec    Weibull;
        } Data;
      };
```

```
/*      ***********************************************************       */
/*      Include File lp_info.h                                             */
/*      ***********************************************************       */
/*                                                                         */
/*      This include file organizes a table for the input                  */
/*      parameters of each logical process, Source, Branch,                */
/*      Queue, Server, and Sink.                                           */
/*                                                                         */
/*      ***********************************************************       */


/* Source date */
struct  Src_Rec {
        long            Num_Gen;                /*  Number of jobs allowed       */
                                                /*  to be generated              */
        int             Out_ID;                 /*  Output process id            */
        struct          Dis_Rec  Dis;           /*  Probability distributions    */
};
 /* Sink data */
struct  Sink_Rec {
        int             Num_In;                 /*  Number of input links        */
};


/* Link data */
struct Link_Rec {
        int             ID;                     /*  Link id                      */
        double          Prob;                   /*  Probability                  */
};


/* Branch data */
struct Branch_Rec {
        int             Num_In;                 /*  Number of input links        */
        int             Num_Out;                /*  Number of output links       */
        struct          Link_Rec Link[MAXLINK]; /*  Records for out links        */
};


/* Queue_Server Data */
struct Q_Srv_Rec {
        int             Out_ID;                 /*  Output process id            */
        int             Q_Size;                 /*  Queue size                   */
        int             Q_Method;               /*  Queue method                 */
        int             Num_In;                 /*  Number of incoming links     */
        struct          Dis_Rec Dis;            /*  Probability distributions    */
};
```

```
union LP_Data_Rec {

        struct Src_Rec        *Src;                    /*    Source data              */
        struct Sink_Rec       *Sink;                   /*    Sink data                */
        struct Branch_Rec     *Branch;                 /*    Branch data              */
        struct Q_Srv_Rec      *Q_Srv;                  /*    Q/Server data            */

} LP_Data[MAXLP];

union LP_Pid_Rec {

        process Source        Src_Pid;                 /*    Source id                */
        process Sink          Sink_Pid;                /*    Sink id                  */
        process Branch        Branch_Pid;              /*    Branch id                */
        process Server        Srv_Pid;                 /*    Server id                */

};

struct All_LP_Rec {

        int                   Type[MAXLP];             /*    Types of LPs             */
        int                   Mach[MAXLP];             /*    Types of Machines        */
        int                   Virt[MAXLP];
        union                 LP_Pid_Rec LP_Pid[MAXLP]; /*   Ids of LPs               */
        process Queue         Q_Pid[MAXLP];            /*    Ids of Queue processes   */

        int                   insock;                  /*    Input socket             */
        int                   outsock;                 /*    Outpit socket            */

        int                   Total_LP;                /*    Number of LPs            */
        long                  Total_Gen;               /*    Number of generated jobs */

        process Collector     Col_Pid;                 /*    Collector process id     */
        process Terminate     Term_Pid;                /*    Terminate process id     */

        double                Sim_Term_Time;           /*    Termination time         */
        double                Stats_Interval;          /*    Stats interval time      */

};
```

```
/*     ***********************************************************         */
/*     Include File lp_param.h                                             */
/*     ***********************************************************         */
/*                                                                         */
/*     This include file declares variables which are sent                 */
/*     by Create process to each type of logical processes.                */
/*                                                                         */
/*     ***********************************************************         */

/* SEND_MSG is a pointer to a transation that takes */
/* a Msg_Rec and returns no value                   */
typedef trans void (*SEND_MSG) (struct Msg_Rec);


struct Src_Param {
        int             id;                     /*  Source id                 */
        int             out_id;                 /*  Output process id         */
        long            num_gen;                /*  Number of jobs allowed    */
                                                /*  to be generated           */
        double          sim_term_time;          /*  Termination time          */
        double          stats_interval;         /*  Stats interval time       */
        struct          Dis_Rec dis;            /*  Probability distributions */
        SEND_MSG        send_msg;               /*  a pointer to a transation */

        trans void      (*send_stats)       (struct Src_Stats_Rec);
};

struct Sink_Param {
        int             id;                     /*  Branch process id         */
        int             num_in;                 /*  Number of input links     */
        double          sim_term_time;          /*  Termination time          */
        double          stats_interval;         /*  Stats interval time       */

        trans void      (*send_stats)       (struct Sink_Stats_Rec);
};

struct Queue_Param {
        int             id;                     /*  Queue process id          */
        int             q_size;                 /*  Queue size                */
        int             q_method;               /*  Queue method              */
        int             num_in;                 /*  Number of input links     */
        double          sim_term_time;          /*  Termination time          */
};

struct Srv_Param {
        int             id;                     /*  Server process id         */
        int             out_id;                 /*  Output process id         */
        double          sim_term_time;          /*  Termination time          */
        double          stats_interval;         /*  Stats interval time       */
        process             Queue que;
        struct              Dis_Rec   dis;              /*   Probability distributions     */
        SEND_MSG            send_msg;

trans void (*send_stats) (struct Q_Srv_Stats_Rec);
};

struct Branch_Param {

        int                 id;                         /*   Branch process id             */
        int                 num_in;                     /*   Number of input links         */
        int                 num_out;                    /*   Number of output links        */
        double              sim_term_time;              /*   Termination time              */
        double              stats_interval;             /*   Stats interval time           */
        struct {

        int                 id;                         /*   Link id                       */
        double              prob;                       /*   Link probability              */
        SEND_MSG            send_msg;
                            } link[MAXLINK];
};

struct Col_Param {

        int                             id;             /*   Colloctor process id          */
        struct All_LP_Rec   All_LP;

};

struct Term_Param {

        int                 id;                         /*   Terminate process id          */

  struct All_LP_Rec   All_LP; };

union LP_Param_Rec {

        struct              Src_Param       src;
        struct              Sink_Param      sink;
        struct              Srv_Param       srv;
        struct              Queue_Param     queue;
        struct              Branch_Param    branch;
        struct              Col_Param       collect;
        struct              Term_Param      term;

} LP_Param;
```

```
/*    **************************************************************         */
/*    Include File lp_stats.h                                                */
/*    **************************************************************         */
/*                                                                           */
/*    This include file declares variables for the statistical               */
/*    report of each process type.                                           */
/*                                                                           */
/*    **************************************************************         */


struct Src_Stats_Rec {

        int             ID;                 /*    Source process id             */
        int             Status;             /*    Status of stats report        */
        int             Num_Left;           /*    Jods have been generated      */
        double          Sim_Time;           /*    Simulation time               */
        double          Ave_Arrival;        /*    Average arrival time          */
        double          Std_Arrival;        /*    Standard arrival time         */
        double          Max_Arrival;        /*    Maximum arrival time          */

};

struct Q_Stats_Rec {

        double          Per_Full;           /*    Percentage of full time       */
        long            Num_In_Q;           /*    Number of jobs in queue       */
        long            Num_Through_Q;      /*    Number of jobs through q      */

};

struct Q_Srv_Stats_Rec {

        int             ID;                 /*    Q/server process id           */
        int             Status;             /*    Status of stats report        */
        double          Sim_Time;           /*    Simulation time               */
        double          Per_Busy;           /*    Percentage of busy time       */
        double          Ave_Service;        /*    Average service time          */
        double          Std_Service;        /*    Standard service time         */
        double          Max_Service;        /*    Maximum service time          */

struct    Q_Stats_Rec    Q_Stats;

};

struct Sink_Stats_Rec {

        int             ID;                 /*    Sink process id               */
        int             Status;             /*    Status of status report       */
        long            Num_Sunk;           /*    Number of sunk jobs           */
        double          Sim_Time;           /*    Simulation time               */

};

struct Col_Stats_Rec {

        long            Interval_num;       /*    Iteration of stats report     */
        int             Status;             /*    Status of stats report        */
```

```
	int			Update;			/*	True, update old report		*/

	   union {
			struct Src_Stats_Rec   *Src_Stats;
			struct Q_Srv_Stats_Rec *Q_Srv_Stats;
			struct Sink_Stats_Rec  *Sink_Stats;
	   } LP_Stats[MAXLP];

	   struct Col_Stats_Rec   *Next;
};

/* From Vopata's thesis[VOPA89]		*/
struct stats_rec {    /*** Stats Structure ***/

	long			num_val;		/*	number of values		*/
	double			max_val;		/*	max value			*/
	double			sum_val;		/*	sum of all values		*/
	double			sum_sq;			/*	sum of squares			*/

};

typedef struct stats_rec STATS;

	void		Stats_Init();		/*	Initalize a Stats Structure		*/
	void		Stats_Val();		/*	Add a value to a Stats Sructure		*/
	double		Stats_Mean();		/*	Return the average of a Stats Structure	*/
	double		Stats_STD();		/*	Return the STD of a Stats Structure
*/
```

```
/*      *****************************************************************
 *      mach.h -- by Edward Vopata
 *      *****************************************************************
 *      List of Machines names for use with c_processor function.
 *      These names will be used to distribute the logical processes.
 *      This is very machine/network dependent.
 *      MAX_MACH indicates the number of possible machines.
 *      WARNING: In order to Distributed Concurrent C to operate properly
 *      all distributed process must be on a compatable machine.
 *      Therefore distribution will be made only on 3b2/3b15's.
 *      ksuvax1 and harris are included for completness.
 *      *****************************************************************
 */


#define    MAX_MACH 19     /*    Number of machines                          */
#define    MAX_VIRT 16     /*    Number of Virtual Processors per Machine    */
#define    MAX_PS 24       /*    Number of Processes per Virtual Processor   */

char *Mach_Name[] = {

/*    Machine No.    */    Machine Name    */    Machine model           */
/*              0    */    "foxtrot",      /*    AT&T 3b2-400     */
/*              1    */    "golf",         /*    AT&T 3b2-400     */
/*              2    */    "hotel",        /*    AT&T 3b2-400     */
/*              3    */    "india",        /*    AT&T 3b2-400     */
/*              4    */    "juliet",       /*    AT&T 3b2-400     */
/*              5    */    "kilo",         /*    AT&T 3b2-400     */
/*              6    */    "lima",         /*    AT&T 3b2-400     */
/*              7    */    "mike",         /*    AT&T 3b2-400     */
/*              8    */    "november",     /*    AT&T 3b2-400     */
/*              9    */    "hack",         /*    AT&T 3b2-400     */
/*             10    */    "alpha",        /*    AT&T 3b2-10      */
/*             11    */    "bravo",        /*    AT&T 3b2-10      */
/*             12    */    "charlie",      /*    AT&T 3b2-10      */
/*             13    */    "delta",        /*    AT&T 3b2-10      */
/*             14    */    "echo",         /*    AT&T 3b2-10      */
/*             15    */    "phobos",       /*    AT&T 3b15        */
/*             16    */    "deimos",       /*    AT&T 3b15        */
/*             17    */    "ksuvax1",      /*    DEC Vax 11/780 */    /* No D-CCC */
/*             18    */    "harris",       /*    Harris HCX 9   */    /* No D-CCC */
};
```

```
/*      *****************************************************************
 *      ksuvax1 & harris were used as development machines.  They are much
 *      faster then the 3b2's. ksuvax1 & harris were not compatable enough
 *      to do distributed process between the two, but they were able to
 *      allow the simulator to be tested on a single processor.
 *      Making debugging much faster, and easier (DBX is a very, very nice
 *      symbolic debugger, while SDB has its drawbacks).
 *      *****************************************************************
 */
```

```
/*      ************************************************************        */
/*      Include File msg.h                                                  */
/*      ************************************************************        */
/*                                                                          */
/*      This include file declares variables for a message,                 */
/*      an incoming link, and an outgoing link.                             */
/*                                                                          */
/*      ************************************************************        */

#define NONE        -1

struct Msg_Rec {

        long            id;             /*  Message id              */
        int             type;           /*  Message type            */
        int             from;           /*  Message sender          */
        int             prior;          /*  Message priority        */
        double          receive_time;   /*  Message receive time    */
        double          send_time;      /*  Message send time       */

};

struct In_Line_Rec {

        double          Time;           /*  Incoming link time      */
        long            Id;             /*  Accepted message id     */
        int             Type;           /*  Accepted message type   */
        int             Caller;         /*  Message sender          */
        int             Selected;       /*  True, accept message    */

};

struct Out_Line_Rec {

        double          Time;           /* Outgoing link time   */
        long            Id;             /* Message id           */
        int             Type;           /* Message type         */
        int             Selected;       /* True, send message   */

};
```

```
/*****************************************************************
 * rand.h -- by Edward Vopata
 *****************************************************************
 * Handle standard C function for generating random numbers.
 * For BSD, the random number generator is random() which generates
 *   a long integer in the range of 0 <= X < 2^31.  The function
 *   srandom() is used to seed the random number generator.
 * For SYS5, the random number generator that corresponds to random()
 *   and random() is called lrand48() & srand48().
 * Mapping the random number generator to rand and the seeding
 *   function to srand will allow different random number generators
 *   to be installed without too much pain.
 * Generating random numbers in the Range of 0 <= X < 1.0 (X is real)
 *   is handled by the function drand01(). This function makes use of
 *   the mapped rand() function.  It may be possible to install a
 *   0 <= X < 1 random number generator to replace drand01().
 * Comments:
 *   random() & lrand48() generate fairly uniform psuedo random
 *     numbers.
 *   If the random number generators are not seeded then they will
 *     produce the same sequence of random numbers on every run.
 *   There are many possible method for generating the seed.
 *     1. (getpid() * getppid()) -- the produce of the process id
 *         and the parent process id is a fairly random number and
 *         makes a good seed value.
 *     2. (time)  -- Some function using the time and date will
 *         also produce a good seed value.
 *    drand01() produces fairly uniform random numbers less
 *      then 1.00.
 *     0.000 values are very, very rare. Disclaimer: This function
 *     may produce values X == 1.00 <E.W.V>
 *****************************************************************
 */

   /* Define SYS5 in Makefile when compiling on SYS 5 systems */
   /* Random number generation functions (standard C functions */
#ifdef SYS5

     long lrand48();           /*     Generate a long X : 0 <= X < 2^31     */
     void srand48();           /*     Seed the random number generator      */

# define rand()                    lrand48()      /*   Map lrand48 to rand          */
# dcfine srand(a)                  srand48(a)     /*   Map srand48 to srand         */

#else /* BSD */

lw(0.1i) lw(1.5i) lw(0.3i) lw(2.0i) l.
                                   long random();    /*   Generate a long X : 0 <= X < 2^31   */
                                   void srandom();   /*   Seed the random number generator    */

#  define rand()       random()      /*   Map random to rand     */
#  define srand(a)     srandom(a)    /*   Map srandom to srand   */
```

```
#endif

   /* provide a real (double) X : 0 <= X < 1 random number */
#define drand01() (((double)(rand() & 0xffffff) / 0xffffff))
```

```
/* spec.h */

process spec Source( )
{
  async trans setup(struct Src_Param);
  trans void term( );
};


process spec Branch( )
{
  async trans setup(struct Branch_Param);
  trans void send_msg(struct Msg_Rec);
  trans void term( );
};


process spec Queue( )
{
  async trans setup(struct Queue_Param);
  trans void send_msg(struct Msg_Rec);
  trans struct Msg_Rec     get_msg( );
  trans struct Q_Stats_Rec get_stats(int);
  trans void term( );
};

process spec Server( )
{
  async trans setup(struct Srv_Param);
  trans void term( );
};

process spec Sink( )
{
  async trans setup(struct Sink_Param);
  trans void send_msg(struct Msg_Rec);
  trans void term( );
};

process spec Terminate( )
{
  async trans setup(struct Term_Param);
  trans void term( );
};

process spec Collector( )
{
  async trans setup(struct Col_Param);
  trans void src_stats(struct Src_Stats_Rec);
  trans void sink_stats(struct Sink_Stats_Rec);
  trans void srv_stats(struct Q_Srv_Stats_Rec);
  trans void term( );
};
```

```
/* Main.cc */

#include "define.h"


main(argc,argv)
int  argc;          /* a count of the number of command line argument */
char **argv;        /* an array of pointer to char                    */
{
   register  i;
   struct All_LP_Rec  *All_LP;
   union  LP_Data_Rec *LP_Data;
   char   filename[50];          /*  the file name for a input data  */


   /* malloc a space to hold a ALL_LP pointer  */
   All_LP = (struct All_LP_Rec *) malloc (sizeof(struct All_LP_Rec));

   /* malloc MAXLP spaces, each one holds a LP_Data pointer */
   LP_Data = (union LP_Data_Rec *)
            malloc (sizeof(union LP_Data_Rec) * MAXLP);

   /* [(] indicates a socket stream is used,
     [n] will be a socket descriptor        */
   if (argv[1][0] == '(')
   {
     sscanf(argv[1],"%d",&All_LP->insock);
     All_LP->outsock = All_LP->insock;
   }
   else
   {
     fprintf(stderr,"0ain, File is used");
     /* prompt the user for a file name which
         will be read as a input data        */
     printf("0nter Input File0);
     gets(filename);
     fprintf(stderr,"55s", filename);
     /* returns a file discriptor or -1 for fail,
         access is 0 for read               */
     if ((All_LP->insock = open(filename,0)) < 0) {
         printf("0pen a file descriptor error");
         c_exit(0);
     }

     /* 1 stands for stdout, the terminal  */
     All_LP->outsock = 1;
   };

   Build_LP(All_LP,LP_Data);

   Create_LP(All_LP,LP_Data);

   for ( i= 0; i < All_LP->Total_LP; i++) {
       switch(All_LP->Type[i]) {
        case SOURCE:
          free ((char *) LP_Data[i].Src);
         break;

        case BRANCH:
          free ((char *) LP_Data[i].Branch);
         break;

       case QUE_SRV:
          free ((char *) LP_Data[i].Q_Srv);
         break;

        case SINK:
          free ((char *) LP_Data[i].Sink);
         break;
      }
   }
free ((char *)LP_Data);
free ((char *)All_LP);
}
```

```
#include "define.h"

void Read_Input(line,sock)
    char   *line;
    int    sock;
    {
         int rval;

       bzero(line, sizeof(line));

       /* Read MAXLINE bytes from sock into a line */
       if ((rval = read(sock, line, MAXLINE)) == -1) {
           fprintf(stderr,"0eading from a socket error");
           exit(1);
       }
        else {
           line[MAXLINE+1] = '  ';
           fprintf(stderr,"0nput line:%s",line);
       }
     }


void Write_Output(line,sock)
    char   *line;
    int    sock;
    {
         int wval;

       /* Write nbytes to sock into a line */
       if (wval = write(sock, line, strlen(line)) == -1)  /* 128 ??? */
         {
           fprintf(stderr,"0riting to a socket error");
           exit(1);
       }
        else
         {
         line[MAXLINE -1] = '  ';
           fprintf(stderr,"0utput line :%s",line);
       }
    }
```

```
/* Build.cc */

#include "define.h"

/*   Example of an Input Model

((990))
((00)(20050.0)80 1001)
((31)8012((20.60)(30.40))
((22)(40050.0)204 1001)
((23)(40050.0)1504 1001)
((24)(40050.0)1405 1002)
((15)1401)
((991))
((980)0 200)                    */

static void Buildup_Dis(dis,ptr)
   struct Dis_Rec     *dis;
   char               *ptr;
   {
     int    dis_type;

     sscanf(ptr,"%d %lf %lf",&dis_type,&dis->Min,&dis->Max);
     switch(dis_type) {

       case FIXED : {
           dis->Dis_Type = FIXED;
         sscanf(ptr,"%*d %*lf %*lf %lf",&dis->Data.Fixed.time);
           break;
       }

       case UNIFORM : {
           dis->Dis_Type = UNIFORM;
         sscanf(ptr,"%*d %*lf %*lf %ld %ld",
                 &dis->Data.Uniform.lower,
                 &dis->Data.Uniform.upper);
           break;
       }

       case POISSON : {
         dis->Dis_Type = POISSON;
         sscanf(ptr,"%*d %*lf %*lf %lf",&dis->Data.Poisson.mean);
           break;
       }

       case BINOMIAL : {
          dis->Dis_Type = BINOMIAL;
        sscanf(ptr,"%*d %*lf %*lf %ld %lf",
                &dis->Data.Binomial.num,
              &dis->Data.Binomial.prob);
```

```
    break;
 }

 case EXPNTL : {
    dis->Dis_Type = EXPNTL;
   sscanf(ptr,"%*d %*lf %*lf %lf",&dis->Data.Expntl.mean);
    break;
 }

 case NORMAL : {
    dis->Dis_Type = NORMAL;
   sscanf(ptr,"%*d %*lf %*lf %lf %lf",
          &dis->Data.Normal.mean,
         &dis->Data.Normal.stdev);
    break;
 }

 case GAMMA : {
    dis->Dis_Type = GAMMA;
   sscanf(ptr,"%*d %*lf %*lf %lf %lf",
          &dis->Data.Gamma.mean,
         &dis->Data.Gamma.k);

    break;
 }

 case BETA  : {
    dis->Dis_Type = BETA;
   sscanf(ptr,"%*d %*lf %*lf %lf %lf",
          &dis->Data.Beta.k1,
       &dis->Data.Beta.k1);
    break;
 }

 case ERLANG :  {
    dis->Dis_Type = ERLANG;
    sscanf(ptr,"%*d %*lf %*lf %lf %ld",
          &dis->Data.Erlang.mean,
       &dis->Data.Erlang.k);
    break;
 }

 case LOGNORMAL : {
    dis->Dis_Type = LOGNORMAL;
    sscanf(ptr,"%*d %*lf %*lf %lf %lf",
         &dis->Data.Lognormal.mean,
       &dis->Data.Lognormal.stdev);
    break;
 }

 case WEIBULL : {
```

```
            dis->Dis_Type = WEIBULL;
            sscanf(ptr,"%*d %*lf %*lf %lf %lf",
               &dis->Data.Weibull.shape,
             &dis->Data.Weibull.scale);
            break;
          }
          default :
           dis->Dis_Type = FIXED;
           dis->Min = 0;
          dis->Max = 0;
           break;

      } /* switch */
    } /* Buildup_Dis  */

/* parse the input data and store the data  */
void Build_LP(All_LP,LP_Data)
    struct  All_LP_Rec   *All_LP;
    union   LP_Data_Rec  *LP_Data;
    {
     register   i;
     int        type, id;
     int        Done = FALSE;
     char        *ptr;
     char        line[MAXLINE + 1];
     void        Read_Input( );
     void        Buildup_Dis( );
     void        Setup_LP( );


     All_LP->Total_LP = 0;
     All_LP->Total_Gen = 0;

     while(!Done) {
       /* Get a line from socket */
       Read_Input(line,All_LP->insock);

       fprintf(stderr,"58s",line);

        ptr = index(line,'(') + 1;
        ptr = index(ptr,'(') + 1;
       sscanf(ptr,"%d %d",&type,&id);

        ptr = index(ptr,')') + 1;
       switch(type) {
         /*<Source> ::= ( ( 0 ID ) ( <Stoch> ) <Mach> <Virt> <Gen> <Out_ID> )
           ( ( 0 0 ) ( 2 0 0 50.0 ) 8 0 100 1 )  */

         case SOURCE : {
             All_LP->Total_LP ++;
             All_LP->Type[id] = SOURCE;
```

```
  /* Malloc a space to hold a Src_Rec struct that the Src
       pointer is point to */
   LP_Data[id].Src = (struct Src_Rec *) malloc (sizeof (struct
                  Src_Rec));
   ptr = index(ptr,'(') + 1;
  /* <Stoch>::= ( <Type> <Min> <Max> <Arg1> [ <Arg2> ] )
   ( 2 0 0 50.0 )   */
   /* pass the address of the field Dis of the struct Src_Rec
       and a pointer points to a char in the input line */
  Buildup_Dis(&LP_Data[id].Src->Dis,ptr);
  ptr = index(ptr,')') + 1;
  sscanf(ptr,"%d %d %ld %d",
          &All_LP->Mach[id],
         &All_LP->Virt[id],
          &LP_Data[id].Src->Num_Gen,
         &LP_Data[id].Src->Out_ID);
  All_LP->Total_Gen += LP_Data[id].Src->Num_Gen;
   break;
 }


/*  <Sink>    ::= ( ( 1 ID ) <Mach> <Virt> <Num_In> )   */
case SINK :       {
   All_LP->Total_LP ++;
   All_LP->Type[id] = SINK;
   LP_Data[id].Sink = (struct Sink_Rec *) malloc (sizeof
                      (struct Sink_Rec));
  sscanf(ptr,"%d %d %d",
        &All_LP->Mach[id],
        &All_LP->Virt[id],
        &LP_Data[id].Sink->Num_In);
   break;
}

/*  <Branch>  ::= ( ( 3 ID ) <Mach> <Virt> <Num_In> <Num_Out>
            ( <Out_list> ) )       */
case BRANCH : {
   All_LP->Total_LP ++;
   All_LP->Type[id] = BRANCH;
   LP_Data[id].Branch = (struct Branch_Rec *) malloc (sizeof
                        (struct Branch_Rec));

  sscanf(ptr,"%d %d %d %d",
        &All_LP->Mach[id],
        &All_LP->Virt[id],
        &LP_Data[id].Branch->Num_In,
        &LP_Data[id].Branch->Num_Out);

  ptr = index(ptr,'(') + 1;
  ptr = index(ptr,'(') + 1;
 for (i = 0; i < LP_Data[id].Branch->Num_Out; i ++) {
        sscanf(ptr,"%d %lf",
              &LP_Data[id].Branch->Link[i].ID,
              &LP_Data[id].Branch->Link[i].Prob);
      if (i < LP_Data[id].Branch->Num_Out - 1) {
          ptr = index(ptr,')') + 1;
          ptr = index(ptr,'(') + 1;
      }
    }
     break;
  }


/* <Q_Server> ::= ( ( 2 ID ) ( <Stoch> ) <Mach> <Virt> <Out_ID>
                <Q_Size> <Q_Method> <Num_In> )   */
case QUE_SRV : {
    All_LP->Total_LP ++;
    All_LP->Type[id] = QUE_SRV;
    LP_Data[id].Q_Srv = (struct Q_Srv_Rec *) malloc (sizeof
                     (struct Q_Srv_Rec));

    ptr = index(ptr,'(') + 1;
  /* <Stoch>::= ( <Type> <Min> <Max> <Arg1> [ <Arg2> ] )
    ( 2 0 0 50.0 )   */
  Buildup_Dis(&LP_Data[id].Q_Srv->Dis,ptr);
  ptr = index(ptr,')') + 1;
  sscanf(ptr,"%d %d %d %d %d %d",
         &All_LP->Mach[id],
         &All_LP->Virt[id],
         &LP_Data[id].Q_Srv->Out_ID,
         &LP_Data[id].Q_Srv->Q_Size,
         &LP_Data[id].Q_Srv->Q_Method,
         &LP_Data[id].Q_Srv->Num_In);
   break;
}

case BEGIN: {
      if (id == 0) { };
  if (id == 1) { };
    break;
}

case START: {
  if (id == 0) {
      Done = TRUE;
    sscanf(ptr,"%lf %lf",
          &All_LP->Sim_Term_Time,
          &All_LP->Stats_Interval);
  }
     break;
}
```

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

  } /* Build_LP */
```

```
/* Create.cc */

static SEND_MSG  Send_Ptr(All_LP,ID)
struct All_LP_Rec   *All_LP;
int   ID;
{

    switch(All_LP->Type[ID]) {

       case BRANCH:
            /* called process.transation name  */
          return All_LP->LP_Pid[ID].Branch_Pid.send_msg;

       case QUE_SRV:
            return All_LP->Q_Pid[ID].send_msg;

       case SINK:
            return All_LP->LP_Pid[ID].Sink_Pid.send_msg;

       default:
          return NULL;
    }
}


void Create_LP(All_LP,LP_Data)
    struct All_LP_Rec   *All_LP;
    union  LP_Data_Rec  *LP_Data;
    {
       register  i;
       register  n;
      char      hostname[10];
      int      hostid;
       char      *program[20];  /*  the path name of a load module on the
                              c_processor  */
#ifdef SYS5
       int      machine[MAXMACH];
#endif
       union    LP_Param_Rec    LP_Param;
      SEND_MSG  Send_Ptr( );



      gethostname(hostname,10);

      for (i = 0; i < MAXMACH; i++) {
        if (strcmp(hostname,Mach_Name[i]) == 0) {
          hostid = i;
        }
      }
```

```
#ifdef SYS5
       for (i = 0; i < All_LP->Total_LP; i++) {
          if (All_LP->Mach[i] != hostid) {
            *program = "LF/a.out";

            machine[All_LP->Mach[i]] =
                 c_processor(Mach_Name[All_LP->Mach[i]],program);
          }
       }
#endif

        for(i = 0; i < All_LP->Total_LP; i++) {
          switch(All_LP->Type[i]) {
            case SOURCE:
             fprintf(stderr, "Oreate, a source process");
             All_LP->LP_Pid[i].Src_Pid = create Source( )
#ifdef SYS5
                              processor(machine[All_LP->Mach[i]])
#endif
             ;
             break;

            case SINK:
             fprintf(stderr,"Oreate, a sink process");
             All_LP->LP_Pid[i].Sink_Pid = create Sink( )
#ifdef SYS5
                              processor(machine[All_LP->Mach[i]])
#endif
              ;
             break;

            case BRANCH:
             fprintf(stderr,"Oreate, a branch process");
             All_LP->LP_Pid[i].Branch_Pid = create Branch( )
#ifdef SYS5
                              processor(machine[All_LP->Mach[i]])
#endif
               ;
               break;

            case QUE_SRV:
             fprintf(stderr, "Oreate, a que process");
             All_LP->LP_Pid[i].Srv_Pid = create Server( )
#ifdef SYS5
                              processor(machine[All_LP->Mach[i]])
#endif
              ;
            fprintf(stderr, "Oreate, a sev process");
            All_LP->Q_Pid[i] = create Queue( )
#ifdef SYS5
                              processor(machine[All_LP->Mach[i]])
#endif
              ;
        break;
    }
}

All_LP->Col_Pid = create Collector( );
 All_LP->Term_Pid = create Terminate( );

 for(i = 0; i < All_LP->Total_LP; i++)
 {
   switch(All_LP->Type[i])
   {
     case SOURCE:
      LP_Param.src.id = i;
      LP_Param.src.out_id = LP_Data[i].Src->Out_ID;
      LP_Param.src.num_gen = LP_Data[i].Src->Num_Gen;
      LP_Param.src.sim_term_time = All_LP->Sim_Term_Time;
      LP_Param.src.stats_interval = All_LP->Stats_Interval;
      LP_Param.src.dis = LP_Data[i].Src->Dis;
      LP_Param.src.send_msg =Send_Ptr(All_LP,LP_Data[i].Src->Out_ID);
         LP_Param.src.send_stats= All_LP->Col_Pid.src_stats;
         /* a tranction call to set up the process */
      All_LP->LP_Pid[i].Src_Pid.setup(LP_Param.src);
       break;

     case SINK:
      LP_Param.sink.id = i;
         LP_Param.sink.num_in = LP_Data[i].Sink->Num_In;
      LP_Param.sink.sim_term_time = All_LP->Sim_Term_Time;
      LP_Param.sink.stats_interval = All_LP->Stats_Interval;
         LP_Param.sink.send_stats = All_LP->Col_Pid.sink_stats;

      All_LP->LP_Pid[i].Sink_Pid.setup(LP_Param.sink);
       break;

     case BRANCH:
      LP_Param.branch.id = i;
      LP_Param.branch.num_in = LP_Data[i].Branch->Num_In;
      LP_Param.branch.num_out = LP_Data[i].Branch->Num_Out;
      LP_Param.branch.sim_term_time = All_LP->Sim_Term_Time;
      LP_Param.branch.stats_interval = All_LP->Stats_Interval;

        for (n = 0; n < LP_Data[i].Branch->Num_Out; n++) {
          LP_Param.branch.link[n].id =
              LP_Data[i].Branch->Link[n].ID;
          LP_Param.branch.link[n].prob =
              LP_Data[i].Branch->Link[n].Prob;
        LP_Param.branch.link[n].send_msg = Send_Ptr(All_LP,
               LP_Data[i].Branch->Link[n].ID);
     };

     All_LP->LP_Pid[i].Branch_Pid.setup(LP_Param.branch);
      break;
```

```
        case QUE_SRV:
         LP_Param.queue.id = i;
           LP_Param.queue.q_size = LP_Data[i].Q_Srv->Q_Size;
           LP_Param.queue.q_method = LP_Data[i].Q_Srv->Q_Method;
           LP_Param.queue.num_in = LP_Data[i].Q_Srv->Num_In;
         LP_Param.queue.sim_term_time = All_LP->Sim_Term_Time;
         All_LP->Q_Pid[i].setup(LP_Param.queue);

         LP_Param.srv.id = i;
           LP_Param.srv.out_id = LP_Data[i].Q_Srv->Out_ID;
         LP_Param.srv.sim_term_time = All_LP->Sim_Term_Time;
         LP_Param.srv.stats_interval = All_LP->Stats_Interval;
         LP_Param.srv.que = All_LP->Q_Pid[i];
           LP_Param.srv.dis = LP_Data[i].Q_Srv->Dis;
         LP_Param.srv.send_msg = Send_Ptr(All_LP,LP_Data[i].Q_Srv->Out_ID);
         LP_Param.srv.send_stats = All_LP->Col_Pid.srv_stats;
         All_LP->LP_Pid[i].Srv_Pid.setup(LP_Param.srv);
          break;

      default:
        fprintf(stderr,"Create:Invalid type (%d)0,All_LP->Type[i]);
          break;
    }
  }

  LP_Param.collect.id = COL_ID;      /* november */
  LP_Param.collect.All_LP = *All_LP;
  All_LP->Col_Pid.setup(LP_Param.collect);

  LP_Param.term.id = TERM_ID;  /* november */
  LP_Param.term.All_LP = *All_LP;
  All_LP->Term_Pid.setup(LP_Param.term);
}
```

```
/* Distrib.cc */

long binomial( );
long poisson( );
long uniform( );
double beta( );
double erlang( );
double expntl( );
double gamma( );
double lognormal( );
double normal( );
double weibull( );


double Get_Time(dis)
struct Dis_Rec  dis;
{
  double  time;

  do {
    switch(dis.Dis_Type) {

      case FIXED: {
          time = dis.Data.Fixed.time;
          break;
      }

      case UNIFORM: {
          time = (double)(uniform(dis.Data.Uniform.lower,
                              dis.Data.Uniform.upper));
        break;
      }

      case POISSON: {
          time = poisson(dis.Data.Poisson.mean);
          break;
      }

      case BINOMIAL: {
          time = (double)(binomial(dis.Data.Binomial.num,
                              dis.Data.Binomial.prob));
        break;
      }

      case EXPNTL: {
          time = expntl(dis.Data.Expntl.mean);
        break;
      }

      case NORMAL: {
          time = normal(dis.Data.Normal.mean,
                    dis.Data.Normal.stdev);
```

```
            break;
        }

        case GAMMA: {
            time = gamma(dis.Data.Gamma.mean,
                    dis.Data.Gamma.k);
            break;
        }

        case BETA: {
            time = beta(dis.Data.Beta.k1,
                    dis.Data.Beta.k1);
            break;
        }

        case ERLANG: {
            time = erlang(dis.Data.Erlang.mean,
                    dis.Data.Erlang.k);
            break;
        }

        case LOGNORMAL: {
            time = (double)(lognormal(dis.Data.Lognormal.mean,
                            dis.Data.Lognormal.stdev));
            break;
        }

        case WEIBULL: {
            time = weibull(dis.Data.Weibull.shape,
                    dis.Data.Weibull.scale);
            break;
        }
      }
  } while (time <= 0.0);  /* it has to be greater than 0.0. Otherwise,
    the source will produce two same time_stamp messages */

  /* Truncate the functions if min or max time > 0) */
  if (dis.Min > 0.0 && time < dis.Min)
    time = dis.Min;

  if (dis.Max > 0.0 && time > dis.Max)
    time = dis.Max;

  return (time);
}
```

```
/*   ***************************************************         *
 *   Stochastic Distribution Functions                           *
 *   These function are from Monte Hall's Thesis [HALL88]        *
 *   ***************************************************         *
*/

/*   ****************************************************        */
/*   Discrete Statistical Distributions                          */
/*   ****************************************************        */

/*   ---INTEGER UNIFORM [a,b] RANDOM VARIATE GENERATOR----       */
/*                                                               */
/*   This function requires two integer bounds as input          */
/*   parameters which represent the range in which the           */
/*   integer random variates are generated.                      */
/*                                                               */
/*   --------------------------------------------------          */


long uniform(lower,upper)
long lower,upper;
{
  long c;

  c = (long) (lower + (upper - lower) * drand01());
  return (c);
}


/*   ---------- POISSON RANDOM VARIATE GENERATOR --------        */
/*                                                               */
/*   This poisson distribution is usually used to model          */
/*   the number of arrivals in a given amount of time.           */
/*   It is related to the exponential function.  The mean        */
/*   is required as an input parameter, and an integer           */
/*   random variate is generated.                                */
/*                                                               */
/*   --------------------------------------------------          */
```

```
long poisson(mean)
double mean;
{
   long n;
   double x,y;

   n = 0;

   if (mean > 6.0) return ((long)normal(mean,sqrt(mean)));
   else {
      y = exp(-1 * mean);
      x = drand01();

      while (x >= y) {
         n = n + 1;
         x = x * drand01();
      }
      return (n);
   }
}


/*    ---------- BINOMIAL RANDOM VARIATE GENERATOR -------              */
/*                                                                      */
/*    According to the SIMSCRIPT book description from                  */
/*    which these functions were borrowed, the binomial                 */
/*    distribution represents the integer number of                     */
/*    successes in n independent trials, each having prob-              */
/*    ability of success p.                                             */
/*                                                                      */
/*    --------------------------------------------------                */

long binomial(num,prob)
long num;
double prob;
{
   register i;
   long sum = 0;

   for (i = 0; i < num; i++)
      if (drand01() <= prob) sum += 1;
   return (sum);
}
```

```
/*     ****************************************************     */
/*     Continuous Statistical Distributions                     */
/*     ****************************************************     */

/*     ----------- BETA RANDOM VARIATE GENERATOR --------        */
/*                                                              */
/*     The input parameters to beta are two variables, which    */
/*     when put together in the formulas below determine the    */
/*     mean (mu) and standard deviation (sd) of the distri-     */
/*     bution:                                                  */
/*                                                              */
/*     mu = k1 / (k1 + k2)                                      */
/*     sd = sqrt((k1 * k2) / (sqr(k1 + k2) * (k1 + k2 + 1)      */
/*                                                              */
/*     ---------------------------------------------            */


double beta(k1,k2)
double k1,k2;
{
   double x;

   x = gamma(k1,k1);
   return (x / (x + gamma(k2,k2)));
}

/*     ---------- ERLANG RANDOM VARIATE GENERATOR -------        */
/*                                                              */
/*     An erlang function is a special case of a gamma          */
/*     function when k is an integer. If k = 1, then the        */
/*     erlang function is the same as the exponential           */
/*     function. The mean (x) and a constant (k) are the        */
/*     input parameters to the function. An extra test was      */
/*     added to this code to assure that the value of the       */
/*     variable e was not equal to zero, primarily so the       */
/*     logarithm function would not be passed a parameter       */
/*     equal to zero.                                           */
/*                                                              */
/*     ---------------------------------------------            */
```

```
double erlang(mean,k)
double mean;
long k;
{
   register i;
   double e;

   do {
      e = 1.0;
      for (i=0; i < k; i++) e *= drand010();
   } while (e == 0.0);
   return (-(mean/k) * log(e));
}

/*     --------- EXPONENTIAL RANDOM VARIATE GENERATOR -----          */
/*                                                                   */
/*     The input parameter for an exponential distribution           */
/*     is the mean (x).  The variance for an exponential             */
/*     distribution is simply the square of the mean.                */
/*                                                                   */
/*     ------------------------------------------------              */


double expntl(mean)
double mean;
{
   double y;

   while ((y = drand010()) == 0.0);
   return ((-mean) * log(y));
}


/*     ----------- GAMMA RANDOM VARIATE GENERATOR --------            */
/*                                                                   */
/*     The gamma function requires a mean (x) and a constant         */
/*     (k) as input parameters.  If k is an integer, then            */
/*     this function is the same as the erlang function.  If         */
/*     k is equal to one, this function is the same as the           */
/*     exponential function.  If k is equal to one-half,             */
/*     this function is the same as the chi-square distri-           */
/*     bution.  The density function for this distribution           */
/*     is given below:                                               */
/*                                                                   */
/*     f(x) = ( (1 / (k-1)! * pow(b,k))  *                           */
/*     pow(x,(k-1)) * exp(-x/b) )                                    */
/*                                                                   */
/*     where the following holds:                                    */
/*     k > 0,  b > 0, and x >= 0                                     */
/*     and the mean is:  x = k * b                                   */
/*     and the variance is:  var = sqr(b) * k                                  */
/*                                                                             */
/*     The gamma function has smaller variance and more                        */
/*     control in parameter selection, and therefore more                      */
/*     realistically represents observed data, such as                         */
/*     service times.  It is often used in preference to the                   */
/*     exponential function, and is closely related to the                     */
/*     beta and erlang functions, according to the SIMSCRIPT                   */
/*     book from which these functions where borrowed.                         */
/*                                                                             */
/*     -------------------------------------------------                       */


double gamma(mean,k)
double mean, k;
{
   double z,a,b,d,e,x,y,w,v;
   long kk;
   register i;

   z = 0.0;
   kk = (long) k;  /* truncation of k */
   d = k - kk;    /* fractional of k */

   if (kk != 0) {
      do {
         e = 1.0;
         for (i=0; i < kk; i++) e *= drand01();
      } while (e == 0.0);
      z = -(log(e));
      if (d == 0.0) return((mean / k) * z);
   }

   a = 1.0 / d;
   b = 1.0 / (1.0 - d);
   y = 2.0;

   while (y > 1.0) {
     x = pow(drand01(),a);
     y = (pow(drand01(),b)) + x;
   }

   w = x / y;
   while ((v = drand01()) == 0.0);
   y = -(log(v));
   return ((w * y + z) * (mean / k));
}


/*     --------- LOG NORMAL RANDOM VARIATE GENERATOR ------                    */
/*                                                                             */
/*    This function requires a mean and standard deviation          */
/*    (sigma) as input parameters. The log normal function          */
/*    is often used to characterize skewed data. The mean           */
/*    and variance of this distribution function are given          */
/*    below:                                                        */
/*                                                                  */
/*    mu = exp(mean + (sqr(sigma) / 2))                             */
/*    sig = exp( (mean * 2) + (sqr(sigma)) ) *                      */
/*    ( (exp (sqr(sigma))) - 1)                                     */
/*                                                                  */
/*    ---------------------------------------------                 */


double lognormal(mean,stdev)
double mean,stdev;
{
  double s,u;

  s = log((stdev * stdev) / (mean * mean) + 1);
  u = log(mean) - (0.5 * s);
  return (double)(exp(normal(u,sqrt(s))));
}


/*    ------------ NORMAL RANDOM VARIATE GENERATOR ------            */
/*                                                                  */
/*    The normal distribution function provides a "bell-            */
/*    shaped curve". It requires the mean (mu) and stan-            */
/*    dard deviation (sigma) as input parameters. If in-            */
/*    appropriate relative values of mean and standard              */
/*    deviation are entered, it is possible that the "tail"         */
/*    of the function can extend into the negative region           */
/*    of the graph (x-axis). This could cause some                  */
/*    complications in regard to generating service times,          */
/*    which have no meaning if negative. An extra test was          */
/*    added to this code to recalculate a new random                */
/*    variate if a variate of less than zero is generated.          */
/*                                                                  */
/*    ---------------------------------------------                 */
```

```
double normal(mean,stdev)
double mean,stdev;
{
  double q,r,s,x,xx,y,yy;

  do {
    s = 2.0;
    while (s > 1.0) {
      x = drand01();
      y = (2.0 * drand01()) - 1;
      xx = x * x;
      yy = y * y;
      s = xx + yy;
    }
    while ((x = drand01()) == 0.0);
    r = sqrt((-2.0) * log(x)) / s;
    q = r * stdev * (xx - yy) + mean;
  } while (q <= 0.0);
  return (q);
}


/*    ----------- WEIBULL RANDOM VARIATE GENERATOR ------          */
/*                                                                */
/*    This function can represent several families of             */
/*    distribution functions depending on the values of the       */
/*    input parameters.  If the shape parameter is equal to       */
/*    one, then this function is the same as the exponen-         */
/*    tial function with a mean equal to the scale para-          */
/*    meter.  There is also a similarity between this             */
/*    function and the gamma distribution when the shape          */
/*    parameter is set equal to two.                              */
/*                                                                */
/*    -----------------------------------------------             */


double weibull(shape,scale)
double shape,scale;
{
  double x;

  while ((x = drand01()) == 0.0);
  return (scale * pow((-log(x)),(1.0 / shape)));
}
```

```
#include "define.h"

/*      *******************************************************************
 *      stats.cc -- by Edward Vopata
 *      *******************************************************************
 *      Function for gather statistical information.  These function use
 *      the Stats structure defined in "stats.h".
 *
 *      Stats_Init -- initialize a Stats struct
 *      Stats_Val  -- add a value to a Stats struct
 *      Stats_Mean -- calculate the average of a Stats struct
 *      Stats_STD  -- calculate the standard deviation of a Stats struct
 *
 *      These function kept track of the number of values added to the
 *      Stats struct, the sum of the values, the sum of the values^2,
 *      and the maximum entered value.
 *      *******************************************************************
 */

/*      *******************************************************************
 *      Function :
 *      Stats_Init()
 *      Parameter :
 *      p - pointer to a STATS structure
 *      Summary :
 *      Initialize the values within the STATS structure.
 *      number of value, sum of the value, and sum of the values square
 *      are assigned 0, max value is assigned -I (a very small value).
 *      *******************************************************************
 */

void Stats_Init(p)       /* Initialize a "stats' structure */
STATS *p;
{

     p->num_val =  0;      /*    number of value <= 0                    */
     p->max_val = -I.0;    /*    max. value <= -I (very small value)
*/
     p->sum_val =  0.0;    /*    sum of the values <= 0                  */
     p->sum_sq  =  0.0;    /*    sum of the values^2 <= 0                */

}

/*      *******************************************************************
 *      Function :
 *      Stats_Val()
 *      Parameter:
 *      p - pointer to a STATS structure
 *      v - floating point value
 *      Summary :
 *      Update a STATS structure with value v.  First check to see if
 *      v is a maxium value and if so, store v in max_val.
 *     update num_val, sum_val, and sum_sq.
 *     *******************************************************************
 */


      void Stats_Val(p,v)                  /*   Add a value to a "stats" structure       */
      STATS  *p;
      double v;
      {
                                           /*   Update the max. value if necessary       */
      if (v > p->max_val) p->max_val = v;
      p->num_val += 1;                     /*   we have another value, increment         */
      p->sum_val += v;                     /*   add the value to the sum                 */
      p->sum_sq  += (v * v);               /*   add the value^2 to sum_sq                */

                                           /*   print the values of the STATS structure  */
                                           /*   Needs to have a Debug Flag               */
/*

   printf("STATS=%X val = %ld, sum_val %lf, sum_sq %lf  max  %lf0,
      p,p->num_val,p->sum_val,p->sum_sq,p->max_val);
*/

}


/*     *******************************************************************
 *     Function :
 *     Stats_Mean()
 *     Parameter:
 *     p - pointer to a STATS structure
 *     Summary :
 *     Calculate the mean (average) of all the values that have been
 *     added to the STATS structure.  If there has been no values added,
 *     then return 0. (Prevents divide by zero errors).
 *     Return:
 *     mean = sum_val / num_val.
 *     *******************************************************************
 */


double Stats_Mean(p)    /* Return the mean value from STATS struct */
STATS *p;
{
    /* calculate and return the average of the Stats struct */
  return (p->num_val != 0) ? p->sum_val/p->num_val : 0;
}


/*     *******************************************************************
 *     Function :
 *     Stats_STD()
*     Parameter:
*     p - pointer to a STATS structure
*     Summary :
*     Calculate the Standard Deviation (STD) of a STATS structure.
*     (Beware of structures that have not had values added to them).
*     Return:
*     STD = square_root( (sum_sq / num_val) - (mean * mean) )
*     *********************************************************************
*/


double Stats_STD(p)
STATS *p;
{
  double avg;  /* Average of a STATS structure */

  if (p->num_val == 0) return 0; /* Check for no values in STATS */
  else
  {
     /* Calculate average of STATS. (could call Stats_Mean()) */
    avg = p->sum_val / p->num_val;
     /* Calculate and return the standard deviation of the
      * Stats struct.  May have problems with negative values.
      */
    return sqrt(p->sum_sq / p->num_val - avg * avg);
  }
}
```

```
#include "define.h"

/*      **************************************************************      */
/*      Process Body Source( )                                              */
/*      **************************************************************      */
/*                                                                          */
/*      This Source process generates a new message based on                */
/*      a user's specified stochastic distribution function and             */
/*      sends the new message to the rest of the modelled system.           */
/*      When the interval time for a statistical report expires,            */
/*      the Source process will send a report to the Collector              */
/*      process.                                                            */
/*                                                                          */
/*      **************************************************************      */


process body Source( )
{
        int             infinite = FALSE;               /*  True, num_gen = 0           */
        int             Timeup = FALSE;                 /*  True, msg >= term_time      */
        int             new_message = FALSE;            /*  True, new message           */
        long            num_msg = 0;                    /*  Msg id                      */
        long            num_stats = 0;                  /*  Interval number             */
        double          inter_arrival;                  /*  Inter_arrival time          */
        double          Time = 0.0;                     /*  Local clock                 */
        double          max_arrival = 0.0;              /*  Maximum inter_arrival       */
        double          Get_Time( );                    /*  Distribution function       */
        STATS           Arvl_Stats;                     /*  Inter_arrival stats         */

        struct          Src_Param Param;                /*  Source parameters           */
        struct          Msg_Rec Msg;                    /*  Message                     */
        struct          Src_Stats_Rec Src_Stats;        /*  Source statistics           */
        struct          In_Line_Rec In_Line;            /*  Incoming line               */
        struct          Out_Line_Rec Out_Line;          /*  Outgoing line               */
        struct          Out_Line_Rec New_Out_Line;      /*  Temporary out line          */
        process         Source Iam;                     /*  Source process id           */


/*      accept the initial specified parameters from the Create.cc              */
        accept setup(source_param) { Param = source_param; };

/*      generates a random seed for a random generator                          */
        srand(getpid( ) * time((long * ) 0));
```

```
In_Line.Id  = -1;
In_Line.Time = 0.0;
In_Line.Type = NULL_MSG;
In_Line.Caller = Param.id;
In_Line.Selected = FALSE;

Out_Line.Id =  -1;
Out_Line.Time = 0.0;
Out_Line.Type = NULL_MSG;
Out_Line.Selected = FALSE;

Stats_Init(&Arvl_Stats);

Msg.id = -1;
Msg.type = NULL_MSG;
Msg.from = NONE;
Msg.prior = 0;
Msg.receive_time = Time;
Msg.send_time = Time;

if (Param.num_gen == 0) { infinite = TRUE; }

/*  get the process id for the Source  */
Iam = (process Source)c_mypid( );

for(;;) {

    if (In_Line.Time == Time)
       In_Line.Selected = TRUE;

     if (Out_Line.Time == Time &&
        In_Line.Time > Out_Line.Time) {
       Out_Line.Selected = TRUE;

      if (new_message) {
         New_Out_Line.Id = Msg.id;
         New_Out_Line.Time  = In_Line.Time;
         New_Out_Line.Type = Msg.type;
      }
      else {
         New_Out_Line.Id = -1;
         New_Out_Line.Time = In_Line.Time;
         New_Out_Line.Type = NULL_MSG;
      };

      if ( (Param.sim_term_time == 0 &&
            Param.num_gen == 0) ||
          (infinite &&
            New_Out_Line.Time == Param.sim_term_time)) {
        fprintf(stderr,
                "0ource[%d], Pass term time",Param.id);
         New_Out_Line.Id = Msg.id;
         New_Out_Line.Time = Param.sim_term_time;
         New_Out_Line.Type = REAL_MSG;
        Timeup = TRUE;
    }
    else if (infinite &&
            New_Out_Line.Time > Param.sim_term_time) {
      fprintf(stderr,
              "0ource[%d], Pass term time",Param.id);
       New_Out_Line.Id = -1;
       New_Out_Line.Time = Param.sim_term_time;
       New_Out_Line.Type = NULL_MSG;
        Timeup = TRUE;
    }
  }

  if (Out_Line.Selected == TRUE) {
   fprintf(stderr,
          "0ource[%d], Out is selected",Param.id);
   Msg.id = New_Out_Line.Id;
    Msg.type = New_Out_Line.Type;
    Msg.send_time = New_Out_Line.Time;
   fprintf(stderr,"0ource[%d], Msg.send_time = %lf",
          Param.id,Msg.send_time);
   fprintf(stderr,"0ource[%d], Msg.id = %d",
          Param.id,New_Out_Line.Id);
   fprintf(stderr,
          "0ource[%d], Before sending out message",
          Param.id);

   (*Param.send_msg)(Msg);

   fprintf(stderr,
          "0ource[%d], After sending out message",
          Param.id);
    Out_Line.Id = New_Out_Line.Id;
    Out_Line.Time = New_Out_Line.Time;
    Out_Line.Type = New_Out_Line.Type;
    Out_Line.Selected = FALSE;
 }

 if (In_Line.Selected == TRUE) {
   fprintf(stderr,
          "0ource[%d], In is selected ",Param.id);
   inter_arrival = Get_Time(Param.dis);
    Stats_Val(&Arvl_Stats,inter_arrival);
    max_arrival = MAX(max_arrival,inter_arrival);
   fprintf(stderr,"0ource[%d], Arrival_time = %lf",
          Param.id,inter_arrival);
   Msg.id = num_msg++;
   Msg.type = REAL_MSG;
   Msg.from = Param.id;
    Msg.receive_time = Time + inter_arrival;
   fprintf(stderr,"0ource[%d], Msg.receive_time = %lf",
          Param.id,Msg.receive_time);

   Param.num_gen--;
```

```
        fprintf(stderr,"0ource[%d], num_gen = %ld",Param.id,
                Param.num_gen);

         In_Line.Id = Msg.id;
         In_Line.Time = Msg.receive_time;
         In_Line.Type = Msg.type;
         In_Line.Selected = FALSE;

         /* phycial source generates a real message  */
        if (In_Line.Type == REAL_MSG)
            new_message = TRUE;
    };

    Time = MIN(In_Line.Time, Out_Line.Time);
    fprintf(stderr,"0ource[%d], Time = %lf",Param.id,Time);

    if (c_transcount(Iam.term) > 0 ||
      (Time == Param.sim_term_time && Timeup == TRUE))
        break;

    /* Statisitcal Output  */
    /* <Source>   ::= ( <ID>< Num_Left> )  */

    if (num_stats != (long)(Time /Param.stats_interval)) {
        num_stats = (long)(Time / Param.stats_interval);
        Src_Stats.ID = Param.id;
        Src_Stats.Status = STATS_NORMAL;
        Src_Stats.Ave_Arrival = Stats_Mean(&Arvl_Stats);
        Src_Stats.Std_Arrival = Stats_STD(&Arvl_Stats);
        Src_Stats.Max_Arrival = max_arrival;
      Src_Stats.Num_Left = Param.num_gen;
        Src_Stats.Sim_Time = Time;
      fprintf(stderr,
              "0ource[%d], Before sending a stats report",
              Param.id);

        (*Param.send_stats)(Src_Stats);
    }
}
if (Time == Param.sim_term_time && Timeup == TRUE) {
   Src_Stats.ID = Param.id;
   Src_Stats.Status = STATS_FINAL;
   Src_Stats.Num_Left = Param.num_gen;
   Src_Stats.Ave_Arrival = Stats_Mean(&Arvl_Stats);
   Src_Stats.Std_Arrival = Stats_STD(&Arvl_Stats);
   Src_Stats.Max_Arrival = max_arrival;
   Src_Stats.Sim_Time = Time;
   fprintf(stderr,"0ource[%d], Before sending a final report",
           Param.id);

   (*Param.send_stats)(Src_Stats);
}
else if (c_transcount(Iam.term) > 0) {
   /* Sending out a negitive time_stamp message */
```

```
        Msg.id = -1;
        Msg.type = TERM_MSG;
        Msg.from = Param.id;
        Msg.receive_time = TERM_TIME;
        Msg.send_time = TERM_TIME;
        fprintf(stderr,
                "0ource[%d], Before sending out a term message",
                Param.id);

        (*Param.send_msg)(Msg);
    }

    fprintf(stderr,"0ource[%d], Ready to terminate",Param.id);
    accept term( ) { };

}
```

```
/*    *************************************************************      */
/*    Process Body Branch( )                                               */
/*    *************************************************************      */
/*                                                                         */
/*    This Branch process receives a job and selects one out of           */
/*    all its outgoing links to send the job. The maximum incoming        */
/*    or outgoing links are limited to 5. The selection of an             */
/*    outgoing link is based on the user's specified link proba-          */
/*    bility instead of the comparison of the time-stamps.                */
/*    Each time the Branch process sends a real message along one         */
/*    of its out links, it has to send a NULL message with the same       */
/*    time-stamp among all the unchosen links. The Branch process         */
/*    does not collect any statistical report in the simulator.           */
/*                                                                         */
/*    *************************************************************      */

process body Branch( )
{

                        register n;               /*  Index                    */
    int                 Out = FALSE;              /*  True, send message       */
    int                 Skip = FALSE;             /*  True, skip to select     */
    int                 Timeup = FALSE;           /*  True, msg >= term time   */
    int                 Done = FALSE;             /*  True, Time >= term time  */
    int                 Stop = FALSE;             /*  True, stop by user       */
    int                 Term = FALSE;             /*  True, terminate          */
    int                 count = 1;                /*  Number of msg            */
    long                Out_Id;                   /*  The last sent out msg    */
    double              Time = 0.0;               /*  Local clock              */
    double              lowest_prob;              /*  Value to choose a link   */
    double              rnd;                      /*  Random value             */

    struct Msg_Rec      Msg;                      /*  Message                  */
    struct Msg_Rec      Temp;                     /*  Message                  */
    struct In_Line_Rec  In_Line[MAXLINK];         /*  Incoming links           */
    struct In_Line_Rec  Smallest[MAXLINK];        /*  Smallest time msg        */
    struct In_Line_Rec  Out_Msg;                  /*  Outgoing message         */
    struct Out_Line_Rec Out_Line[MAXLINK];        /*  Outgoing links           */
    struct Out_Line_Rec New_Out_Line;             /*  temporary links          */
    struct Branch_Param Param;                    /*  Branch parameters        */
    process Branch      Iam;                      /*  Branch process id        */

/*  accept the initial parameters form the Create Process                  */
    accept setup(branch_param)  {Param = branch_param; };

/*  get a random seed for the random number generater                      */
    srand(getpid( ) * time((long * ) 0));
```

```
for (n = 0; n < Param.num_in; n ++)
{
  In_Line[n].Id = -1;
  In_Line[n].Caller = NONE;
  In_Line[n].Selected = FALSE;
}

for (n = 0; n < Param.num_out; n ++)
{
  Out_Line[n].Id = -1;
  Out_Line[n].Time = 0.0;
  Out_Line[n].Type = NULL_MSG;
  Out_Line[n].Selected = FALSE;
}

Out_Id = -1;

Iam = (process Branch)c_mypid( );

for(;;)
  select {
     (!Done && !Stop && !Term):
        /* Selection */
           /* Find out the smallest message */
           Smallest[count] = In_Line[0];
        fprintf(stderr,"0ranch, Smallest.type = %d",
                  Smallest[count].Type);
        fprintf(stderr,"0ranch, Smallest.time = %lf",
                  Smallest[count].Time);

         for (n = 0; n < Param.num_in - 1; n++) {
          if (Smallest[count].Time == In_Line[n+1].Time) {
               if (Smallest[count].Type == NULL_MSG)
                 Smallest[count] = In_Line[n+1];
            else if(In_Line[n+1].Type != NULL_MSG)
                 Smallest[count+1] = In_Line[n+1];
          }
          else if (Smallest[count].Time > In_Line[n+1].Time)
               Smallest[count] = In_Line[n+1];

        }

        if (Smallest[count].Type == TERM_MSG) {
          fprintf(stderr,"0ranch, received a TERM_MSG");

             while (count < Param.num_in) {
               accept send_msg(Job)
                 suchthat(Job.from !=
                              Smallest[count].Caller) {
             Temp = Job;
          if (Temp.type == TERM_MSG)
            count++;
      }
    }
      Stop = TRUE;
      Time = TERM_STAMP;
}

/* Select the next input lines which have the
      same time stamp with the current simulation time */
  for (n = 0; n < Param.num_in; n++)
    if (In_Line[n].Time == Time)
      {
      In_Line[n].Selected = TRUE;
    fprintf(stderr,
              "0ranch, In_Line[%d] is selected",n);
      };


for (n = 1; n <= count; n++) {
  Out_Msg = Smallest[n];

    if (!Skip) {
    rnd = drand01();
    lowest_prob = 0.0;
  }


  for (n = 0; n < Param.num_out; n++) {

      if (Out_Line[n].Time == Time &&
          Out_Msg.Time > Out_Line[n].Time) {

      if (lowest_prob <= rnd &&
          rnd < Param.link[n].prob + lowest_prob) {
            Out_Line[n].Selected = TRUE;
            Out = TRUE;
            Skip = FALSE;
        break;
      }
      lowest_prob += Param.link[n].prob;
    }
   else
          Skip = TRUE;
  }

  if (Out == TRUE) {

    if (Out_Msg.Id != Out_Id) {
        New_Out_Line.Id = Out_Msg.Id;
        New_Out_Line.Time = Out_Msg.Time;
      New_Out_Line.Type = REAL_MSG;
}
   else {
      New_Out_Line.Id = -1;
     New_Out_Line.Time = Out_Msg.Time;
      New_Out_Line.Type = NULL_MSG;
};

if (Param.sim_term_time != 0) {

       if (Out_Msg.Time == Param.sim_term_time) {
         New_Out_Line.Id = Out_Msg.Id;
        New_Out_Line.Time = Param.sim_term_time;
        New_Out_Line.Type = Out_Msg.Type;
           Timeup = TRUE;
  }
      else if (Out_Msg.Time >
              Param.sim_term_time) {
         New_Out_Line.Id = -1;
        New_Out_Line.Time = Param.sim_term_time;
        New_Out_Line.Type = NULL_MSG;
           Timeup = TRUE;
  }
}

  for (n = 0; n < Param.num_out; n++) {

     if (Out_Line[n].Selected == TRUE) {
         Msg.id = New_Out_Line.Id;
         Msg.type = New_Out_Line.Type;
     Msg.from = Param.id;
        Msg.receive_time = Out_Msg.Time;
     Msg.send_time = New_Out_Line.Time;
  }
     else {
         Msg.id = -1 ;
         Msg.type = NULL_MSG;
     Msg.from = Param.id;
        Msg.receive_time = Out_Msg.Time;
     Msg.send_time = New_Out_Line.Time;
  }

 fprintf(stderr,
            "0ranch, Msg.id = %d",Msg.id);
 fprintf(stderr,
            "0ranch, msg.send_time = %lf",
            Msg.send_time);
 fprintf(stderr,
            "0ranch, Before sending a msg");
 (*Param.link[n].send_msg)(Msg);
 fprintf(stderr,
                    "0ranch, After sending a msg");

        Out_Id = New_Out_Line.Id;
        Out_Line[n].Id = Msg.id;
          Out_Line[n].Time = Msg.send_time;
          Out_Line[n].Type = Msg.type;
          Out_Line[n].Selected = FALSE;
      }
    }
   };

   Out = FALSE;

    for (n = 0; n < Param.num_in; n++) {

      if (In_Line[n].Selected == TRUE) {
      fprintf(stderr,
                  "0ranch, Before accepting a msg");
        accept send_msg(Job)
            suchthat((In_Line[n].Caller == NONE &&
                   (Job.id == 0 || Job.id == -1)) ||
                    Job.from == In_Line[n].Caller)
            by (Job.send_time) {
         Msg = Job;
         fprintf(stderr,
                    "0ranch, Accepted msg.id = %d",
                    Msg.id);
         fprintf(stderr,
                    "0ranch, Msg.send_time = %lf",
                    Msg.send_time);
            In_Line[n].Id = Msg.id;
         In_Line[n].Time = Msg.send_time;
             In_Line[n].Type = Msg.type;

         if (In_Line[n].Caller == NONE)
            In_Line[n].Caller = Msg.from;

             In_Line[n].Selected = FALSE;
      }
    }
}

 /* compute Time*/
   Time = New_Out_Line.Time;

if (Time != Param.sim_term_time)
for (n = 0; n < Param.num_in; n ++)
    if (In_Line[n].Time < Time)
       Time = In_Line[n].Time;
fprintf(stderr, "0ranch, Time = %lf",Time);
```

```
            if (Time == Param.sim_term_time && Timeup == TRUE)
              Done = TRUE;

      or
        (Done && !Term):
            while (c_transcount(Iam.send_msg) > 0)
              accept send_msg(Job) {  };

            accept term( ) {
              Term = TRUE;
            }

      or
        (Stop && !Term):
          for (n = 0; n < Param.num_out; n++) {
            Msg.id = -1 ;
            Msg.type = TERM_MSG;
           Msg.from = Param.id;
           Msg.receive_time = TERM_TIME;
           Msg.send_time = TERM_TIME;

           fprintf(stderr,
                   "0ranch, Before sending a term msg");
           (*Param.link[n].send_msg)(Msg);
           fprintf(stderr,
                   "0ranch, After sending a term msg");
         }

         accept term( ) {
           fprintf(stderr,"0ranch, accepted term call");
            Term = TRUE;
          }

     or
       (Term):
         terminate;
    }
}
```

```
#include "define.h"

/*    ***********************************************************     */
/*    Process Body Queue ( )                                          */
/*    ***********************************************************     */
/*                                                                    */
/*    This Queue process accepts a job and stores it into its queue   */
/*    until its associated server requests the job.                   */
/*    The functions of each queue can be divided into three:          */
/*    1.) As long as the queue is not full, the queue process can     */
/*    select the smallest time-stamped job(s) and put it into         */
/*    its queue. If the numbers of the smallest time-stamped          */
/*    jobs are greater than the available spaces in the queue,        */
/*    the extra jobs will be stored in a temporary queue. Once        */
/*    a space is available in the queue, jobs stored in the           */
/*    temporary queue will be moved immediately in a FIFO order.      */
/*    2.) As long as the queue is not empty, the queue process is     */
/*    ready to accept a "job request" from its server. Based on       */
/*    a user-specified method, such as FIFO, the queue process        */
/*    sends out the desired job and adjusts its queue to be           */
/*    ready for accepting a next request. The current local time      */
/*    will be the time-stamp of the job sent out.                     */
/*    3.) Whenever its server requests a statistical report, the      */
/*    queue process will calculate the necessary information          */
/*    and give it to the server.                                      */
/*                                                                    */
/*    ***********************************************************     */

process body Queue( )
{
        register        n;                          /*  Index                        */
        int             timeup = FALSE;             /*  True, msg >= term_time       */
        int             put_in_q = FALSE;           /*  True, queue msg              */
        int             Stop = FALSE;               /*  True, stop by user           */
        int             Done = FALSE;               /*  True, Time >= term_time      */
        int             Term = FALSE;               /*  True, terminate              */
        int             through_q = 0;              /*  Number through Q             */
        int             current_size = 0;           /*  Current queue size           */
        int             temp_size = 0;              /*  Temporary queue size         */
        int             fifo = 1;                   /*  FIFO message                 */
        int             lifo = 1;                   /*  LIFO message                 */
        int             siro = 1;                   /*  SIRO message                 */
        int             prior = 1;                  /*  Prior message                */
        int             count = 1;                  /*  Num of smallest_time msg     */
        int             num = 1;                    /*  Num of smallest_time msg     */
        int             status;                     /*  Status of stats report       */
        double          Time = 0.0;                 /*  Local clock                  */

        struct          Msg_Rec Msg;                /*  Message                      */
        struct          Msg_Rec queue[MAXSIZE+1];   /*  Message queue                */
        struct          Msg_Rec Temp[MAXSIZE+1];    /*  Temporary msg queue
```

```
*/
        struct          Msg_Rec New_Msg;                  /*   Accepted message          */
        struct          Msg_Rec Last_In_Q;                /*   Last message in queue     */
        struct          Queue_Param Param;                /*   Queue parameter           */
        struct          Q_Stats_Rec Q_Stats;              /*   Queue Statistics          */
        struct          In_Line_Rec In_Line[MAXLINK];     /*   Incoming links            */
        struct          In_Line_Rec Smallest[MAXLINK];    /*   Outgoing links            */
        process         Queue Iam;                        /*   Queue process id          */
```

```
accept setup(queue_param) { Param = queue_param; };
fprintf(stderr,"0ueue[%d], Q_Size =%d",Param.id,Param.q_size);
fprintf(stderr,"0ueue[%d], Q_method =%d",Param.id,Param.q_method);

for (n = 1; n <= Param.q_size + I; n ++) {
  queue[n].id = -1;   /* messages stored in the Queue */
  queue[n].type = NULL_MSG;
  queue[n].from = NONE;
  queue[n].prior = 0;
  queue[n].receive_time = Time;
  queue[n].send_time = Time;
}

for (n = 0; n < Param.num_in; n ++) {
  In_Line[n].Id = -1;
  In_Line[n].Time = 0.0;
  In_Line[n].Type = NULL_MSG;
  In_Line[n].Caller = NONE;
  In_Line[n].Selected = FALSE;
}

Last_In_Q.id = -I;   /* messages stored in the Queue */
Last_In_Q.type = NULL_MSG;
Last_In_Q.from = NONE;
Last_In_Q.prior = 0;
Last_In_Q.receive_time = Time;
Last_In_Q.send_time = Time;

Iam = (process Queue)c_mypid( );


for(;;)
  select {
      (!Done && !Stop && !Term && current_size < Param.q_size && !timeup):
        /* Selection */
           Smallest[num] = In_Line[0];
        fprintf(stderr,"0ueue[%d], Smallest msg_type = %d",
                  Param.id,Smallest[num].Type);
        fprintf(stderr,"0ueue[%d], Smallest msg_time = %.4f",
                  Param.id,Smallest[num].Time);

         for (n = 0; n < Param.num_in - I; n++) {
          if (Smallest[num].Time == In_Line[n+1].Time) {
               if (Smallest[num].Type == NULL_MSG)
                  Smallest[num] = In_Line[n+I];
            else if(In_Line[n+1].Type != NULL_MSG)
                  Smallest[num+I] = In_Line[n+1];
          }
          else if (Smallest[num].Time > In_Line[n+1].Time)
               Smallest[num] = In_Line[n+I];
        }

        if (Smallest[num].Type == TERM_MSG) {
          fprintf(stderr,"0ueue[%d],received TERM_MSG",Param.id);
      while (c_transcount(Iam.send_msg) > 0)
        accept send_msg(Job)
          suchthat(Job.from != Smallest[count].Caller)

      Stop = TRUE;
      Time = TERM_STAMP;
}

  for (n = 0; n < Param.num_in; n++) {
    if (In_Line[n].Time == Time) {
      In_Line[n].Selected = TRUE;
     fprintf(stderr,"0ueue[%d], In[%d] is selected",
            Param.id,n);
  }
}

if (num > Param.q_size - current_size) {
     temp_size = num - (Param.q_size - current_size);

     for (n = 1; n <= temp_size; n++) {
    Temp[n].id = Smallest[n+temp_size].Id;
    Temp[n].type = Smallest[n+temp_size].Type;
    Temp[n].from = Smallest[n+temp_size].Caller;
    queue[n].prior = 0;
    queue[n].receive_time = Smallest[n+temp_size].Time;
  }
  num = num - temp_size;
}
  else {
    for(n = 1; n <= num; n++) {

      if (Last_In_Q.receive_time == Time &&
         Smallest[n].Time > Last_In_Q.receive_time ) {
      fprintf(stderr,"0ueue[%d], current size < q_size",
               Param.id);

      put_in_q = TRUE;
      if (Smallest[n].Id != Last_In_Q.id) {
           New_Msg.id = Smallest[n].Id;
        New_Msg.type = REAL_MSG;
        New_Msg.from = Param.id;
           New_Msg.prior = Msg.prior;
        New_Msg.receive_time = Smallest[n].Time;
      }
        else {
           New_Msg.id = -1;
        New_Msg.type = NULL_MSG;
        New_Msg.from = Param.id;
           New_Msg.prior = 0;
        New_Msg.receive_time = Smallest[n].Time;
      }

      if (Param.sim_term_time != 0) {
           if (Smallest[n].Time == Param.sim_term_time) {
              New_Msg.id = Smallest[n].Id;
            New_Msg.type = Smallest[n].Type;
            New_Msg.from = Param.id;
              New_Msg.prior = 0;
            New_Msg.receive_time = Param.sim_term_time;
               timeup = TRUE;
          }
            else if(Smallest[n].Time > Param.sim_term_time) {
            New_Msg.id = -1;
          New_Msg.type = NULL_MSG;
          New_Msg.from = Param.id;
            New_Msg.prior = 0;
          New_Msg.receive_time = Param.sim_term_time;
             timeup = TRUE;
          }
        }

        if (put_in_q == TRUE) {
           current_size++;
          queue[current_size] = New_Msg;
            Last_In_Q = queue[current_size];
          fprintf(stderr,"0ueue[%d],[%d].id = %d",
              Param.id,current_size,queue[current_size].id);
          fprintf(stderr,"0ueue[%d], current_size = %d",
                   Param.id,current_size);
            put_in_q = FALSE;
        }
      }
    }
  };

   for (n = 0; n < Param.num_in; n++) {
     if (In_Line[n].Selected == TRUE) {
    fprintf(stderr,"0ueue[%d], before accepting a message",
             Param.id);
    accept send_msg(Job)
         suchthat((In_Line[n].Caller == NONE &&
                 (Job.id == 0 || Job.id == -1)) ||
                Job.from == In_Line[n].Caller)
         by (Job.send_time) {
      Msg = Job;
      fprintf(stderr,"0ueue[%d], Accepted msg_id =%ld",
                Param.id, Msg.id);
      fprintf(stderr,"0ueue[%d], Msg.send_time = %.4f",
                Param.id, Msg.send_time);
         In_Line[n].Id = Msg.id;
      In_Line[n].Time = Msg.send_time;
         In_Line[n].Type = Msg.type;
      if (In_Line[n].Caller == NONE)
          In_Line[n].Caller = Msg.from;
         In_Line[n].Selected = FALSE;
       }
    }
  }
  /* compute Time*/
    n = 0;
 Time = Last_In_Q.receive_time;
 fprintf(stderr,"0ueue[%d], Time = %.4f", Param.id,Time);

   while (n < Param.num_in) {
     if (In_Line[n].Time < Time)
         Time = In_Line[n].Time;
       n++;
 };
```

or

```
  (!Done && !Stop && !Term && current_size > 0):

      accept get_msg( )
            suchthat(c_transcount(Iam.get_msg) > 0) {

      switch(Param.q_method) {
          case FIFO :
                Msg.id = queue[fifo].id;
                 Msg.type = queue[fifo].type;
                 Msg.from = Param.id;
                 Msg.receive_time = queue[fifo].receive_time;
                 Msg.send_time = Time;
                queue[fifo].id = -5;
                break;

        case LIFO :
              for (n = 1; n < current_size; n++)
               {
                 if (queue[lifo].receive_time ==
                      queue[n+1].receive_time)
                      lifo = n+1 ;
               else
                        break;
                 }
              Msg.id = queue[lifo].id;
              Msg.type = queue[lifo].type;
              Msg.from = Param.id;
                 Msg.receive_time = queue[lifo].receive_time;
                 Msg.send_time = Time;
               queue[lifo].id = -5;
                 break;

       case SIRO :
             for (n = 1; n < current_size; n++)
                {
                  if (queue[n].receive_time ==
                     queue[n+1].receive_time)
                     count++;
             else
                        break;
           }
```

```
                siro = rand() % count;
              Msg.id = queue[siro].id;
              Msg.type = queue[siro].type;
              Msg.from = Param.id;
                Msg.receive_time = queue[siro].receive_time;
                Msg.send_time = Time;
               queue[siro].id = -5;
               break;

        case PRIO :
               for (n = 1; n < current_size; n++)
               {
                 if (queue[n].receive_time ==
                      queue[n+1].receive_time
                     || queue[n].prior < queue[n+1].prior)
                   prior = n+1;
             }
              Msg.id = queue[prior].id;
              Msg.type = queue[prior].type;
              Msg.from = Param.id;
                Msg.receive_time = queue[prior].receive_time;
                Msg.send_time = Time;
               queue[prior].id = -5;
               break;
    }
    fprintf(stderr,"0ueue[%d], Msg send time = %.4f",
                  Param.id,Msg.send_time);
    fprintf(stderr,"0ueue[%d], Msg_id = %d",
                  Param.id,Msg.id);
    treturn(Msg);
  };

  for (n = 1; n <= current_size; n++) {
     if (queue[n].id == -5) {
         if (n == current_size) {
         queue[n].id = -1;
         queue[n].type = NULL_MSG;
         queue[n].from = NONE;
         queue[n].prior = 0;
         queue[n].receive_time = 0.0 ;
         queue[n].send_time = 0.0;
       }
         else {
          queue[n] = queue[n+1];
           queue[n+1].id = -5;
       }
    }
  };

  through_q++;
   current_size--;

  if (temp_size > 0) {
      current_size++;
```

```
            queue[current_size] = Temp[1];
            Last_In_Q = queue[current_size];

            for(n = 1; n <= temp_size; n++)
              Temp[n] = Temp[n++];

            temp_size--;
        };

        if (timeup && current_size == 0 && temp_size == 0 ||
             Msg.send_time == Param.sim_term_time)
            Done = TRUE;

  or

          /*  <ID> <Per_Full> <In_Q> <Through_Q> )  */
      (!Done && !Stop && !Term):
        accept get_stats(stat)
              suchthat (c_transcount(Iam.get_stats) > 0) {
            Q_Stats.Per_Full = (double)((100 * current_size)
                                /Param.q_size);
            Q_Stats.Num_In_Q = current_size;
            Q_Stats.Num_Through_Q = through_q;
          treturn(Q_Stats);
      }

  or
      (Done && !Term):
        accept get_stats(stat) {
            status = stat;
            Q_Stats.Per_Full = (double)(100 * current_size)
                                /Param.q_size;
            Q_Stats.Num_In_Q = current_size;
            Q_Stats.Num_Through_Q = through_q;
          treturn(Q_Stats);
      }
        if (status == STATS_FINAL) {
          accept term( )
              { Term = TRUE; }
      }

  or
      (Stop && !Term):
          for(;;)
            select {
              accept get_msg( ) {
                  Msg.id = -1;
                  Msg.type = TERM_MSG;
                  Msg.from = Param.id;
                    Msg.receive_time = TERM_TIME;
                    Msg.send_time = TERM_TIME;
                    treturn(Msg);
              }
          or
```

```
                    accept get_stats(stat) {
                        treturn(Q_Stats);
                      }
              or
                    accept term( ) {
                    fprintf(stderr,"Queue[%d], accepted term call",
                              Param.id);
                    if (c_transcount(Iam.get_stats) > 0)
                         accept get_stats(stat) {
                              treturn(Q_Stats);
                            }

                        Term = TRUE;
                        break;
                    }

              }

      or (Term):
          terminate;
   }
 }
```

```
#include "define.h"

/*     ************************************************************     */
/*     Process Body Server( )                                           */
/*     ************************************************************     */
/*                                                                      */
/*     This Server process gets a job from its queue, generates a       */
/*     service time based on a user specified probability               */
/*     distribution, adds the service time onto the original time-      */
/*     stamp of the job and sends it out.                               */
/*     The service time is the lookahead time for the server process.   */
/*     Based on the assumption of the algorithm, increasing accepted    */
/*     message time will not decrease senting message time. The         */
/*     Source will guarantee a reasonable service time to each job.     */
/*     When the interval time for a statistical report expires, the     */
/*     server process requests information from its queue process,      */
/*     incorporates its own, and sends it out.                          */
/*                                                                      */
/*     ************************************************************     */

process body Server( )
{
     register      i;                               /*   Index                    */
     int           Timeup = FALSE;                  /*   True, msg >= term_time   */
     int           Done = FALSE;                    /*   True, Time >= term_time  */
     int           Stop = FALSE;                    /*   True, stop by user       */
     int           Term = FALSE;                    /*   True, terminate          */
     int           size = 0;                        /*   Size of stored message   */
     long          num_stats = 0;                   /*   Interval number          */
     double        service_time = 0.0;              /*   Service time             */
     double        sum_service_time = 0.0;          /*   Sum service time         */
     double        Predict = 0.0;                   /*   Predict time             */
     double        Time = 0.0;                      /*   Local clock              */
     double        Get_Time( );                     /*   Distribution function    */
     STATS         Service_Stats;                   /*   Server time stats        */

     struct        Srv_Param Param;                 /*   Server parameters        */
     struct        Msg_Rec Msg;                     /*   Message                  */
     struct        Msg_Rec Stored_Msg[3];           /*   Queued message           */
     struct        Q_Stats_Rec Q_Stats;             /*   Queue statistics         */
     struct        Q_Srv_Stats_Rec Q_Srv_Stats;     /*   Q/Server statistics
*/
     struct        In_Line_Rec In_Line;             /*   Incoming link            */
     struct        Out_Line_Rec Out_Line;           /*   Outgoing link            */
     struct        Out_Line_Rec New_Out_Line;       /*   Temporary outgoing link  */
     process       Server Iam;                      /*   Server process id        */
```

```
accept setup(server_param) { Param = server_param; };

srand(getpid( ) * time((long * ) 0));


In_Line.Id = -1;
In_Line.Time = 0.0;
In_Line.Type = NULL_MSG;
In_Line.Caller = NONE;
In_Line.Selected = FALSE;

Out_Line.Id = -1;
Out_Line.Time = 0.0;
Out_Line.Type = NULL_MSG;
Out_Line.Selected = FALSE;

Msg.id = -1;
Msg.type = NULL;
Msg.from = Param.id;
Msg.receive_time = Time;
Msg.send_time = Time;

for (i = 0; i < 3; i ++) {
  Stored_Msg[i].id = -1;
  Stored_Msg[i].type = NULL;
  Stored_Msg[i].from = Param.id;
  Stored_Msg[i].receive_time = Time;
  Stored_Msg[i].send_time = Time;
}

Stats_Init(&Service_Stats);

Iam = (process Server)c_mypid( );


for(;;)
  select {
      (!Done && !Stop && !Term):

       if (In_Line.Time != 0.0) {
          service_time = Get_Time(Param.dis);
             Stats_Val(&Service_Stats,service_time);
          fprintf(stderr,"0erver[%d], server time = %lf",
                    Param.id,service_time);
            Predict = In_Line.Time + service_time;

             while (Predict < Out_Line.Time) {
            service_time = Get_Time(Param.dis);
              Predict = In_Line.Time + service_time;
          }
             sum_service_time += service_time;
          fprintf(stderr,"0erver[%d], predict = %lf",
                    Param.id,Predict);
       }
```

```
 if (In_Line.Time == Time)
     In_Line.Selected = TRUE;

 if (Out_Line.Time == Time && Predict > Out_Line.Time) {
     Out_Line.Selected = TRUE;
   fprintf(stderr,"0erver[%d], Out is selected",Param.id);

   if (size > 0) {
       New_Out_Line.Id = Stored_Msg[0].id;
       New_Out_Line.Time = Stored_Msg[0].send_time;
       New_Out_Line.Type = Stored_Msg[0].type;
         size--;
         Stored_Msg[0] = Stored_Msg[1];
         Stored_Msg[1] = Stored_Msg[2];
     Stored_Msg[3].id = -1;
     Stored_Msg[3].type = NULL;
     Stored_Msg[3].from = Param.id;
     Stored_Msg[3].receive_time = 0.0;
     Stored_Msg[3].send_time = 0.0;

      ]
     else if (In_Line.Id != Out_Line.Id) {
      New_Out_Line.Id = In_Line.Id;
      New_Out_Line.Time = Predict;
      New_Out_Line.Type = In_Line.Type;
   ]
  else {
     New_Out_Line.Id = -1;
     New_Out_Line.Time = Predict;
     New_Out_Line.Type = NULL_MSG;
  };

   if ( Param.sim_term_time != 0) {
       if (New_Out_Line.Time == Param.sim_term_time) {
         New_Out_Line.Id = In_Line.Id;
         New_Out_Line.Time = Param.sim_term_time;
         New_Out_Line.Type = REAL_MSG;
           Timeup = TRUE;
     }
       else if (New_Out_Line.Time > Param.sim_term_time) {
         New_Out_Line.Id = -1;
         New_Out_Line.Time = Param.sim_term_time;
         New_Out_Line.Type = NULL_MSG;
           Timeup = TRUE;
     ]
   }
}
 if (Out_Line.Selected == FALSE && In_Line.Id != Out_Line.Id) {
     size++;
     Stored_Msg[size] = Msg;
]

 if (Out_Line.Selected == TRUE) {
    Msg.id = New_Out_Line.Id;
    Msg.type = New_Out_Line.Type;
    Msg.from = Param.id;
    Msg.receive_time = Msg.send_time;
     Msg.send_time = New_Out_Line.Time;

    fprintf(stderr,"0erver[%d],Msg.send_time =%lf",
            Param.id,Msg.send_time);
    fprintf(stderr,"0erver[%d], Msg id = %d",Param.id,Msg.id);
    fprintf(stderr,"0erver[%d], Before sending msg",Param.id);
    (*Param.send_msg)(Msg);
    fprintf(stderr,"0erver[%d], After sending msg",Param.id);

     Out_Line.Id = New_Out_Line.Id;
     Out_Line.Time = New_Out_Line.Time;
     Out_Line.Type = New_Out_Line.Type;
      Out_Line.Selected = FALSE;
  }

 if (In_Line.Selected == TRUE)
  {
   fprintf(stderr,"0erver[%d], Before getting msg",Param.id);
     Msg = Param.que.get_msg( );
   fprintf(stderr,"0erver[%d], Msg.receive_time = %lf",
            Param.id,Msg.send_time);
   fprintf(stderr,"0erver[%d], Msg id = %d",Param.id,Msg.id);

   if (Msg.type == TERM_MSG) {
        Stop = TRUE;
        In_Line.Selected = FALSE;
   }
      else {
       In_Line.Id = Msg.id;
       In_Line.Time = Msg.send_time;
       In_Line.Type = Msg.type;
        In_Line.Caller = Msg.from;
        In_Line.Selected = FALSE;
      }
 }

Time = MIN(In_Line.Time, Out_Line.Time);
fprintf(stderr,"0erver[%d], Time = %lf",Param.id, Time);

if (Time == Param.sim_term_time && Timeup == TRUE)
      Done = TRUE;

 /* Statisitcal Output  */
 /* Q_Server> ::= <ID> <Per_Busy> */
if (!Done) {
 if (num_stats != (long)(Time / Param.stats_interval)) {
      num_stats = (long)(Time / Param.stats_interval);

      Q_Stats = Param.que.get_stats(STATS_NORMAL);
      Q_Srv_Stats.ID = Param.id;
```

```
            Q_Srv_Stats.Status = STATS_NORMAL;
            Q_Srv_Stats.Sim_Time = Time;
            Q_Srv_Stats.Per_Busy = (Q_Srv_Stats.Sim_Time) ?
                (double) (100 * sum_service_time) / Time :0;
            Q_Srv_Stats.Ave_Service = Stats_Mean(&Service_Stats);
            Q_Srv_Stats.Std_Service = Stats_STD(&Service_Stats);
            Q_Srv_Stats.Max_Service = Service_Stats.max_val;
            Q_Srv_Stats.Q_Stats.Per_Full = Q_Stats.Per_Full;
            Q_Srv_Stats.Q_Stats.Num_In_Q = Q_Stats.Num_In_Q;
            Q_Srv_Stats.Q_Stats.Num_Through_Q = Q_Stats.Num_Through_Q;

          fprintf(stderr,"0erver[%d], before sending a stats",
                   Param.id);
           (*Param.send_stats)(Q_Srv_Stats);
         }
        };

or
   (Done && !Term):
         fprintf(stderr,"0erver[%d], ready to send a final stats",
                  Param.id);
         Q_Stats = Param.que.get_stats(STATS_FINAL);

         Q_Srv_Stats.ID = Param.id;
         Q_Srv_Stats.Status = STATS_FINAL;
         Q_Srv_Stats.Sim_Time = Time;
           Q_Srv_Stats.Per_Busy = (Q_Srv_Stats.Sim_Time) ?
               (double) (100 * sum_service_time) / Time :0;
           Q_Srv_Stats.Ave_Service = Stats_Mean(&Service_Stats);
           Q_Srv_Stats.Std_Service = Stats_STD(&Service_Stats);
           Q_Srv_Stats.Max_Service = Service_Stats.max_val;
         Q_Srv_Stats.Q_Stats.Per_Full = Q_Stats.Per_Full;
         Q_Srv_Stats.Q_Stats.Num_In_Q = Q_Stats.Num_In_Q;
         Q_Srv_Stats.Q_Stats.Num_Through_Q = Q_Stats.Num_Through_Q;

         fprintf(stderr,"0erver[%d], Before send a final stats",
                  Param.id);
         (*Param.send_stats)(Q_Srv_Stats);
         fprintf(stderr,"0erver[%d], After send a final stats",
                  Param.id);

         accept term( ) {
           fprintf(stderr,"0erver[%d], accepted term call",
                    Param.id);
             Term = TRUE;
           }

or
   (Stop && !Term):
    Msg.id = -1;
    Msg.type = TERM_MSG;
    Msg.from = Param.id;
    Msg.receive_time = TERM_TIME;
      Msg.send_time = TERM_TIME;
```

```
            fprintf(stderr,"0erver[%d], Before send Term_Msg",Param.id);
            (*Param.send_msg)(Msg);

             accept term( )
             {  Term = TRUE; }

        or (Term):
            terminate;
        }
    }
```

```
#include "define.h"

/*    ************************************************************       */
/*    Process Body Sink( )                                               */
/*    ************************************************************       */
/*                                                                       */
/*    This Sink process selectes a job with the smallest                 */
/*    time-stamp among all incoming links and destroys it.               */
/*    When the interval time for a statistical report expires,           */
/*    the sink process will send a report to the Collector               */
/*    process.                                                           */
/*                                                                       */
/*    ************************************************************       */


process body Sink( )
{
        register    n = 0;                          /*  Index                    */
        int         count = 1;                      /*  Index                    */
        int         Timeup = FALSE;                 /*  True, msg >= term-time   */
        int         Done = FALSE;                   /*  True, Time >= term-term  */
        int         Stop = FALSE;                   /*  True, stop by user       */
        int         Term = FALSE;                   /*  Ture, terminate          */
        long        num_statis = 0;                 /*  Interval number          */
        long        num_sunk = 0;                   /*  Sunk_job number          */
        double      Time = 0.0;                     /*  Local clock              */

        struct      Msg_Rec Msg;                    /*  Message                  */
        struct      Msg_Rec Temp;                   /*  Message                  */
        struct      Out_Line_Rec Sunk;              /*  Sunk record              */
        struct      Out_Line_Rec New_Sunk;          /*  Temporary Sunk record    */
        struct      Sink_Param Param;               /*  Sink parameters          */
        struct      Sink_Stats_Rec Sink_Stats;      /*  Sink statistical report  */
        process     Sink Iam;                       /*  Sink process id          */
        struct      In_Line_Rec In_Line[MAXLINK];   /*  Incoming link            */
        struct      In_Line_Rec Smallest[MAXLINK];  /*  Smallest time-           */
                                                    /*  stampted message         */
```

```
accept setup(sink_param) { Param = sink_param; };

for (n = 0; n < Param.num_in; n ++) {
  In_Line[n].Id = -1;
  In_Line[n].Time = 0.0;
  In_Line[n].Type = NULL_MSG;
  In_Line[n].Caller = NONE;
  In_Line[n].Selected = FALSE;
}

Sunk.Id = -1;
Sunk.Time = 0.0;
Sunk.Type = NULL_MSG;
Sunk.Selected = FALSE;

Iam = (process Sink)c_mypid( );


for(;;)
  select {
       (!Done && !Stop):
           count = 1;
           Smallest[count] = In_Line[0];
        fprintf(stderr, "0ink, Smallest.msg_type =%d",
                  Smallest[count].Type);
        fprintf(stderr, "0ink, Smallest.Time =%lf",
                  Smallest[count].Time);

       /* Select the smallest time-stamped
             incoming message(s)           */
         for (n = 0; n < Param.num_in - 1; n++) {
         if (Smallest[count].Time == In_Line[n+1].Time) {
             if (Smallest[count].Type == NULL_MSG)
                 Smallest[count] = In_Line[n+1];
           else if(In_Line[n+1].Type != NULL_MSG)
                 Smallest[count+1] = In_Line[n+1];
         }
         else if (Smallest[count].Time > In_Line[n+1].Time)
               Smallest[count] = In_Line[n+1];
       }

       /*     Accept a TERM_MSG          */
       if (Smallest[count].Type == TERM_MSG) {
         fprintf(stderr,"0ink,Accepted TERM_MSG");

           while (count < Param.num_in) {
             accept send_msg(Job)
           suchthat(Job.from != Smallest[count].Caller) {
              Temp = Job;

                 if (Temp.type == TERM_MSG)
                   count++;
           }
        }
```

```
   Stop = TRUE;
    Time = TERM_STAMP;
}

  for (n = 0; n < Param.num_in; n++)
    if (In_Line[n].Time == Time) {
      In_Line[n].Selected = TRUE;
     fprintf(stderr,
              "0ink, In_Line[%d] is selected",n);
  };


for(n = 1; n <= count; n++) {
  if (Sunk.Time == Time &&
       Smallest[n].Time > Sunk.Time) {
    Sunk.Selected = TRUE;
    fprinif(stderr,"0ink, Sunk is selected");

    if (Smallest[n].Type == REAL_MSG) [
         New_Sunk.Id = Smallest[n].Id;
      New_Sunk.Time = Smallest[n].Time;
         New_Sunk.Type = Smallest[n].Type;
    }
    else {
         New_Sunk.Id = -1;
      New_Sunk.Time = Smallest[n].Time;
         New_Sunk.Type = NULL_MSG;
    }

      if (Param.sim_term_time != 0) {
        if (Smallest[n].Time ==
             Param.sim_term_time) {
           New_Sunk.Id = Smallest[n].Id;
           New_Sunk.Time = Param.sim_term_time;
           New_Sunk.Type = Smallest[n].Type;
            Timeup = TRUE;
     }
        else if (Smallest[n].Time >
                Param.sim_term_time) [
           New_Sunk.Id = -1;
           New_Sunk.Time =Param.sim_term_time;
           New_Sunk.Type = NULL_MSG;
            Timeup = TRUE;
     }
    }
  }
}
  if (Sunk.Selected == TRUE) {
    if (New_Sunk.Type == REAL_MSG ) {
      num_sunk++;
    fprinif(stderr,
              "0ink, Num_sunk = %ld",num_sunk);
  }
```

```
        Sunk.Id = New_Sunk.Id;
      Sunk.Time = New_Sunk.Time;
        Sunk.Type = New_Sunk.Type;
        Sunk.Selected = FALSE;
    }

  for (n = 0; n < Param.num_in; n++) {
     if (In_Line[n].Selected == TRUE) {
       accept send_msg(Job)
           suchthat((In_Line[n].Caller == NONE &&
                     (Job.id == 0 || Job.id == -1)) ||
                    Job.from == In_Line[n].Caller)
           by (Job.send_time) {
            Msg = Job;
         fprintf(stderr,
              "0ink, Accepted Msg id = %ld",Msg.id);
          In_Line[n].Time = Msg.send_time;
          In_Line[n].Id = Msg.id;
            In_Line[n].Type = Msg.type;
           In_Line[n].Selected = FALSE;

            if (In_Line[n].Caller == NONE)
              In_Line[n].Caller = Msg.from;
     }
    }
  };

   /* Compute local time*/
  Time = Sunk.Time;
  for (n = 0; n < Param.num_in; n ++) {
     if (Time > In_Line[n].Time)
         Time = In_Line[n].Time;
  };
  fprintf(stderr,"0ink, Time = %lf",Time);

  if (Time == Param.sim_term_time && Timeup)
      Done = TRUE;

  if (!Done) {
     /* Statisitcal Output  */
    if (num_statis !=
         (long)(Time / Param.stats_interval)) {
        num_statis =
             (long)(Time /Param.stats_interval);
        Sink_Stats.ID = Param.id;
          Sink_Stats.Status = STATS_NORMAL;
        Sink_Stats.Num_Sunk = num_sunk;
      Sink_Stats.Sim_Time = Time;
        (*Param.send_stats)(Sink_Stats);
    }
 };

or
  (Done && !Term):
```

```
            Sink_Stats.ID = Param.id;
            Sink_Stats.Status = STATS_FINAL;
            Sink_Stats.Num_Sunk = num_sunk;
            Sink_Stats.Sim_Time = Time;
            fprintf(stderr, "0ink, before send a final report");
            (*Param.send_stats)(Sink_Stats);

             accept term( ) {
              fprintf(stderr,"0ink, accepted term call");
                Term = TRUE;
            }

        or
          (Stop && !Term):
            accept term( );
                Term = TRUE;
        or
         (Term):
            terminate;
  }

 }
```

```
/*     ************************************************************     */
/*     Process Body Collector( )                                        */
/*     ************************************************************     */
/*                                                                      */
/*     This Collector process collects a complete statistical report    */
/*     at each specified time interval and sends it to the user. The    */
/*     format of the statistical report is shown in Vopata's thesis     */
/*     [VOPA88].                                                        */
/*     Each process, except Branch processes, sends a statistical       */
/*     report at each specified time interval to the Collector          */
/*     process based on its local time. Because the local time of       */
/*     each process is not updated by a fixed constant value,           */
/*     sometimes, report(s) might be skipped. In order not to skip      */
/*     too many reports, once the Collector process recognizes that     */
/*     one or more reports are skipped from a process, the previous     */
/*     report sent by that process would be used to fill in the         */
/*     skipped iteration.                                               */
/*     The following two conditions would cause the Collector process   */
/*     to activate the Terminate process.                               */
/*     1.) After sending a statistical report to the user, if a         */
/*     "termination simulation" control message results, the           */
/*     Collector process would initiate the Terminate process.          */
/*     2.) If one of the processes passes the termination time and     */
/*     is ready to terminate, the process uses a final statistical      */
/*     report to notify the Collector process. When the Collector       */
/*     recognizes that final statistical reports have been sent by     */
/*     all the necessary processes, it initiates the Terminate          */
/*     process.                                                         */
/*                                                                      */
/*     ************************************************************     */

/*     -------------------------------------------------------          */
/*     Mallocate a new statistical report                               */
/*     -------------------------------------------------------          */

static struct Col_Stats_Rec *   Init_Record(Param,num)
struct Col_Param    *Param;
long   num;
{
   register    i;
   struct Col_Stats_Rec   *ptr;

   ptr = (struct Col_Stats_Rec *) malloc
            (sizeof (struct Col_Stats_Rec));
   ptr->Interval_num = num;
   ptr->Update = FALSE;
```

```
  for (i = 0; i < Param->All_LP.Total_LP; i++) {
    if (Param->All_LP.Type[i] != BRANCH)
      switch(Param->All_LP.Type[i]) {
         case SOURCE:
           ptr->LP_Stats[i].Src_Stats = NULL;
           break;

        case QUE_SRV:
          ptr->LP_Stats[i].Q_Srv_Stats = NULL;
           break;

         case SINK:
           ptr->LP_Stats[i].Sink_Stats = NULL;
          break;

         default:
           break;
      }
  }
  ptr->Next = NULL;
  return (ptr);
}


/*   ------------------------------------------------------------                    */
/*   Get a pointer which points to a desired report                                  */
/*   ------------------------------------------------------------                    */

static struct Col_Stats_Rec *   Get_Ptr(Param,head,num)
struct Col_Param      *Param;
struct Col_Stats_Rec  *head;
long num;
{
  register      i;
  struct Col_Stats_Rec   *ptr;

  ptr = head;

  while (ptr->Next != NULL && num >= ptr->Next->Interval_num)
     ptr = ptr->Next;

  if (num > ptr->Interval_num)
   for (i = ptr->Interval_num + 1; i <= num ; i ++) {
      ptr->Next = Init_Record(Param,i);
      ptr = ptr->Next;
    }
  return(ptr);
}
```

```
/*   --------------------------------------------------------             */
/*   Return 'true' if a complete statistical report is received          */
/*   --------------------------------------------------------             */


static int Is_Complete(Param,head)
struct Col_Param      *Param;
struct Col_Stats_Rec   *head;
{
   register    i;
   struct Col_Stats_Rec  *ptr;
   int    Finished = TRUE;

   ptr = head;

   for ( i= 0; i < Param->All_LP.Total_LP; i++) {
     if (Param->All_LP.Type[i] != BRANCH)
       switch(Param->All_LP.Type[i]) {
          case SOURCE:
            if (ptr->LP_Stats[i].Src_Stats == NULL)
              Finished = FALSE;
              break;

        case QUE_SRV:
          if (ptr->LP_Stats[i].Q_Srv_Stats == NULL)
              Finished = FALSE;
              break;

         case SINK:
           if (ptr->LP_Stats[i].Sink_Stats == NULL)
             Finished = FALSE;
            break;
       }
        if (!Finished)
          break;
   }
   return(Finished);
}


/*   --------------------------------------------------------             */
/*   Return 'true' if a complete final report is received                */
/*   --------------------------------------------------------             */
```

```
static int Is_Done(Param, head)
struct Col_Param      *Param;
struct Col_Stats_Rec   *head;
{
   register   i;
   struct Col_Stats_Rec  *ptr;
   int   All_Done = TRUE;
   ptr = head;
   for (i = 0; i < Param->All_LP.Total_LP; i++) {
    if (Param->All_LP.Type[i] != BRANCH)
      switch(Param->All_LP.Type[i]) {
        case SOURCE:
          if (ptr->LP_Stats[i].Src_Stats->Status != STATS_FINAL)
             All_Done = FALSE;
              break;
        case QUE_SRV:
          if (ptr->LP_Stats[i].Q_Srv_Stats->Status != STATS_FINAL)
              All_Done = FALSE;
              break;

         case SINK:
           if (ptr->LP_Stats[i].Sink_Stats->Status != STATS_FINAL)
             All_Done = FALSE;
             break;
      }
     if (!All_Done)
        break;
   }
  return(All_Done);
};


/*    ----------------------------------------------------                                      */
/*    Send a complete statistical report to the Front-End                                      */
/*    ----------------------------------------------------                                      */
```

```
static void Send_Stats(Param, head)
struct Col_Param      *Param;
struct Col_Stats_Rec  *head;
{
   register  i;
   int       sock;
   char      line[MAXLINE+1];

   sock = Param->All_LP.outsock;

   sprintf(line,"(%.4lf)",
          Param->All_LP.Stats_Interval * head->Interval_num);

   Write_Output(line,sock);

   for(i = 0; i < Param->All_LP.Total_LP; i++) {
    if (Param->All_LP.Type[i] != BRANCH) {
     switch(Param->All_LP.Type[i]) {
      case SOURCE:
         sprintf(line,"(%d %d)", i,
           head->LP_Stats[i].Src_Stats->Num_Left);
         break;

       case QUE_SRV:
        sprintf(line,"(%d %.4lf %.4lf %.4ld %.4ld)", i,
           head->LP_Stats[i].Q_Srv_Stats->Per_Busy,
           head->LP_Stats[i].Q_Srv_Stats->Q_Stats.Per_Full,
           head->LP_Stats[i].Q_Srv_Stats->Q_Stats.Num_In_Q,
           head->LP_Stats[i].Q_Srv_Stats->Q_Stats.Num_Through_Q);
         break;

       case SINK:
         sprintf(line,"(%d %d)", i,
         head->LP_Stats[i].Sink_Stats->Num_Sunk);
           break;

      default:
           break;
     }
       Write_Output(line,sock);
    }
   }
  sprintf(line,"$$");
  Write_Output(line,sock);
}


/*    ------------------------------------------------------                    */
/*    Use a standard I/O to send a statistical report                                    */
/*    ----------------------------------------------------                               */


static void Send_File(Param, head, final_num)
struct Col_Param       *Param;
struct Col_Stats_Rec   *head;
long  final_num;
{
  register   i;

  printf("                                        ");
  printf("0==============================================");
  printf("    Interval Number : (%ld)",head->Interval_num);
  printf("    Interval Time   : (%.4lf)",
        Param->All_LP.Stats_Interval * head->Interval_num);
  if (head->Interval_num == final_num)
  printf("            Final Report"    );
  printf("0==============================================");
```

```
for(i = 0; i < Param->All_LP.Total_LP; i++) {
 if (Param->All_LP.Type[i] != BRANCH) {
  switch(Param->All_LP.Type[i]) {
    case SOURCE:
     printf("0Source]: %d  Sim-Time: %lf",
          i,
           head->LP_Stats[i].Src_Stats->Sim_Time);
      printf("  { Inter Arrival Time } ");
      printf("     Ave : %.4lf ",
          head->LP_Stats[i].Src_Stats->Ave_Arrival);
      printf("     STD : %.4lf ",
          head->LP_Stats[i].Src_Stats->Std_Arrival);
      printf("     Max : %.4lf ",
          head->LP_Stats[i].Src_Stats->Max_Arrival);
      printf("  { Num Left }: %ld",
          head->LP_Stats[i].Src_Stats->Num_Left);
     printf("0--------------------------------");
      break;

    case QUE_SRV:
    printf("0Q/Server]: %d      Simulation: %.4lf",
       i,
       head->LP_Stats[i].Q_Srv_Stats->Sim_Time);
    printf("0ueue: { Full }: %.4lf",
       head->LP_Stats[i].Q_Srv_Stats->Q_Stats.Per_Full);
    printf("      { Num through Q }: %ld",
       head->LP_Stats[i].Q_Srv_Stats->Q_Stats.Num_Through_Q);
    printf("      { Num In Q }: %ld",
       head->LP_Stats[i].Q_Srv_Stats->Q_Stats.Num_In_Q);

    printf("0erver: { Busy }: %.4lf",
        head->LP_Stats[i].Q_Srv_Stats->Per_Busy);

     printf("      { Service Time } : ");
     printf("         Ave : %.4lf ",
       head->LP_Stats[i].Q_Srv_Stats->Ave_Service);
     printf("         STD : %.4lf ",
      head->LP_Stats[i].Q_Srv_Stats->Std_Service);
     printf("         Max : %.4lf ",
       head->LP_Stats[i].Q_Srv_Stats->Max_Service);
    printf("0-----------------------------------");
     break;

    case SINK:
    printf("0Sink]: %d      Simulation: %.4lf",
       i,
        head->LP_Stats[i].Sink_Stats->Sim_Time);

    printf(" { Number Sunk }: ");
     printf("116d",
              head->LP_Stats[i].Sink_Stats->Num_Sunk);
        printf("0--------------------------------------");
          break;

        default:
          break;
      }
    }
  }
}


/*      -----------------------------------------------------                    */
/*      Frees a pointer pointing to a report which has been sent out            */
/*      -----------------------------------------------------                    */
```

```
static void Free_Record(Param,head)
struct Col_Param        *Param;
struct Col_Stats_Rec    **head;
{
  register  i;
  long   num;
  struct Col_Stats_Rec   *ptr;

  ptr = *head;
  num = (*head)->Interval_num;

  if ((*head)->Next != NULL)
    *head = (*head)->Next;
  else {
    num++;
    *head = Init_Record(Param,num);
  }


  ptr->Next = NULL;

  for ( i= 0; i < Param->All_LP.Total_LP; i++) {
    if (Param->All_LP.Type[i] != BRANCH)
      switch(Param->All_LP.Type[i]) {
          case SOURCE:
            if (ptr->LP_Stats[i].Src_Stats != NULL)
            free ((char *) ptr->LP_Stats[i].Src_Stats);
            break;

        case QUE_SRV:
          if (ptr->LP_Stats[i].Q_Srv_Stats != NULL)
            free ((char *) ptr->LP_Stats[i].Q_Srv_Stats);
            break;

          case SINK:
            if (ptr->LP_Stats[i].Sink_Stats != NULL)
            free ((char *) ptr->LP_Stats[i].Sink_Stats);
           break;
      }
  }
  free((char *)ptr);
}


/*    ------------------------------------------------------------          */
/*    Waits a result from a user after sending out a report                 */
/*    ------------------------------------------------------------          */
```

```
static int Wait_Response(Param)
struct Col_Param   *Param;
{
  int  sock;
  char *ptr;
  int  signal1, signal2;
  char line[MAXLINE+1];
  char  filename[50];

  sock = Param->All_LP.insock;

  Read_Input(line,sock);
  fprintf(stderr,"0ollector, command: %s",line);
  ptr = index(line,'(') + 1;
  ptr = index(ptr,'(') + 1;
  sscanf(ptr,"%d %d",&signal1,&signal2);

  switch(signal1)
  {
   case CONT:
      fprintf(stderr,"0ollector, sig = continue");
      return(TRUE);

   case TERM:
      fprintf(stderr,"0ollector, sig = term");
       Param->All_LP.Term_Pid.term( );
      return(FALSE);

   default:
      fprintf(stderr,"0ollector, Invalid Command ");
       return(TRUE);
  }
}


process body Collector( ) {

register    i;                     /*   Index                         */
int         ID;                    /*   Index                         */
int         Done = FALSE;          /*   True, complete final          */
int         Continue = TRUE;       /*   True, continue                */
int         Term = FALSE;          /*   True, terminate               */
int         Complete = FALSE;      /*   True, a complete report       */
char        line[MAXLINE];         /*   The output line               */
long        final_num = 0;         /*   A final report index          */
long        src_num[MAXLINE];      /*   Source interval num           */
long        sink_num = 0;          /*   Sink interval num             */
long        srv_num[MAXLINE];      /*   Server interval num           */
long      src_oldnum[MAXLINE];    /*   Source previous num          */
long      sink_oldnum = 0;        /*   Source previous num          */
long      srv_oldnum[MAXLINE];    /*   Server previous num          */
long      old_num_sunk = 0;       /*   Sink data                    */
long      old_num_left            /*   Sink data                    */
long      num = 1;                /*   Initial interval num         */
long      old_in_q = 0;           /*   Queue data                   */
long      old_through_q = 0;      /*   Queue data                   */
double    old_per_busy = 0.0;     /*   Server data                  */
double    old_per_full = 0.0;     /*   Queue data                   */


struct   Col_Param         *Param;        /*   Collector paramter          */
struct   Src_Stats_Rec     pre_src;       /*   Previous Source report      */
struct   Q_Srv_Stats_Rec   pre_q_srv;     /*   Previous Q/Server report    */
struct   Sink_Stats_Rec    pre_sink;      /*   Previous Sink report        */
struct   Src_Stats_Rec     src;           /*   Source report               */
struct   Q_Srv_Stats_Rec   q_srv;         /*   Q/Server report             */
struct   Sink_Stats_Rec    sink;          /*   Sink report                 */
struct   Col_Stats_Rec     *head, *ptr;   /*   pointer                     */
```

```
Param = (struct Col_Param *) malloc (sizeof(struct Col_Param));
accept setup(col_param) { (*Param) = col_param; };
old_num_left = Param->All_LP.Total_Gen;
head = Init_Record(Param,num);

for (i = 0; i < MAXLINE; i++) {
    src_num[i] = 0;
    srv_num[i] = 0;
    src_oldnum[i] = 0;
    srv_oldnum[i] = 0;
}

for(;;)
  select {
      (Continue && !Complete && !Done && !Term):
     /*  <Source>    ::= ( <ID>< Num_Left> ) */
      accept src_stats (src_rec) {
       src = src_rec;
        ID = src.ID;
       fprintf(stderr,"0ollector, accept source[%d] stats",
               src.ID);
        src_num[ID] = (long)(src.Sim_Time /
                    Param->All_LP.Stats_Interval);
      fprintf(stderr,"0ollector, source interval = %ld",
                src_num[ID]);
      if (final_num == 0 && src.Status == STATS_FINAL)
           final_num = src_num[ID];

      if (src.Status == STATS_FINAL) {
          ptr = Get_Ptr(Param,head,final_num);
        fprintf(stderr,"0ollector, source final");
      }
       else
        ptr = Get_Ptr(Param,head,src_num[ID]);

        if (ptr->LP_Stats[src.ID].Src_Stats == NULL) {
            ptr->LP_Stats[src.ID].Src_Stats =
                    (struct Src_Stats_Rec *)
                   malloc (sizeof(struct Src_Stats_Rec));
        }
```

```
         else
            ptr->Update = TRUE;

            *(ptr->LP_Stats[src.ID].Src_Stats) = src;

         if (src_num[ID] - src_oldnum[ID] >= 2) {
            src_oldnum[ID]++;

             for (i = src_oldnum[ID]; i < src_num[ID]; i++) {
            ptr = Get_Ptr(Param,head,i);

            if (ptr->LP_Stats[src.ID].Src_Stats == NULL) {
                  ptr->LP_Stats[src.ID].Src_Stats =
                       (struct Src_Stats_Rec *)
                        malloc (sizeof(struct Src_Stats_Rec));
            }
            else
                  ptr->Update = TRUE;

            pre_src.ID = src.ID;
               pre_src.Num_Left = old_num_left;

               *(ptr->LP_Stats[src.ID].Src_Stats) = pre_src;
             }
         }
        src_oldnum[ID] = src_num[ID];
        pre_src.Num_Left = src.Num_Left;
           Complete = Is_Complete(Param,head);
        fprintf(stderr,"0ollector, complete = %d",Complete);

        if (Complete && head->Interval_num == final_num) {
           Done = Is_Done(Param,head);
              if (!Done)
                 Complete = FALSE;
        }
     }
or
    /* <Sink>     ::= ( <ID> <Num_Sunk> ) */
     accept sink_stats (sink_rec) {
      fprintf(stderr,"0ollector, accept sink stats");
      sink = sink_rec;

        sink_num = (long)(sink.Sim_Time/
                  Param->All_LP.Stats_Interval);

      if (final_num == 0 && sink.Status == STATS_FINAL)
         final_num = sink_num;

         if (sink.Status == STATS_FINAL)
            ptr = Get_Ptr(Param,head,final_num);
```

```
          else
               ptr = Get_Ptr(Param,head,sink_num);

          if (ptr->LP_Stats[sink.ID].Sink_Stats == NULL) {
               ptr->LP_Stats[sink.ID].Sink_Stats =
                    (struct Sink_Stats_Rec *) malloc
                      (sizeof(struct Sink_Stats_Rec));
          }
          else
              ptr->Update = TRUE;

          *(ptr->LP_Stats[sink.ID].Sink_Stats) = sink;

          if (sink_num - sink_oldnum >= 2)  {
               sink_oldnum++;
               for (i = sink_oldnum; i < sink_num; i++)
           {
            ptr = Get_Ptr(Param,head,i);

            if (ptr->LP_Stats[sink.ID].Sink_Stats == NULL) {
                 ptr->LP_Stats[sink.ID].Sink_Stats =
                      (struct Sink_Stats_Rec *)
                      malloc (sizeof(struct Sink_Stats_Rec));
            }
            else
                 ptr->Update = TRUE;

            pre_sink.ID = sink.ID;
               pre_sink.Num_Sunk = old_num_sunk;

            *(ptr->LP_Stats[sink.ID].Sink_Stats) = pre_sink;
            }
          }
           sink_oldnum = sink_num;
          old_num_sunk = sink.Num_Sunk;
          Complete = Is_Complete(Param,head);
          fprintf(stderr,"0ollector, complete = %d",Complete);

          if (Complete && head->Interval_num == final_num) {
             Done = Is_Done(Param,head);
              if (!Done)
                Complete = FALSE;
          }
     }

or
    /* <Q_Server> ::= ( <ID> <Per_Busy> <Per_Full> <In_Q>
                        <Through_Q> )               */
     accept srv_stats (q_srv_rec) {
      q_srv = q_srv_rec;
       ID = q_srv.ID;
```

```
fprintf(stderr,"0ollector, accept q_server[%d] stats",
        q_srv.ID);
 srv_num[ID] = (long)(q_srv.Sim_Time/
       Param->All_LP.Stats_Interval);
fprintf(stderr,"0ollector, num = %ld",srv_num[ID]);

if (final_num == 0 && q_srv.Status == STATS_FINAL)
    final_num = srv_num[ID];

if (q_srv.Status == STATS_FINAL) {
   fprintf(stderr,"0ollector, q_srv STATS_FINAL");
    ptr = Get_Ptr(Param,head,final_num);
}
 else
      ptr = Get_Ptr(Param,head,srv_num[ID]);

if (ptr->LP_Stats[q_srv.ID].Q_Srv_Stats == NULL)
    ptr->LP_Stats[q_srv.ID].Q_Srv_Stats =
         (struct Q_Srv_Stats_Rec *) malloc
         (sizeof(struct Q_Srv_Stats_Rec));
else
    ptr->Update = TRUE;

 *(ptr->LP_Stats[q_srv.ID].Q_Srv_Stats) = q_srv;

 if (srv_num[ID] - srv_oldnum[ID] >= 2) {
   srv_oldnum[ID]++;

    for(i = srv_oldnum[ID]; i < srv_num[ID]; i++) {
      ptr = Get_Ptr(Param,head,i);

   if (ptr->LP_Stats[q_srv.ID].Q_Srv_Stats == NULL) {
            ptr->LP_Stats[q_srv.ID].Q_Srv_Stats =
            (struct Q_Srv_Stats_Rec *) malloc
             (sizeof(struct Q_Srv_Stats_Rec));
   }
   else
        ptr->Update = TRUE;

   pre_q_srv.ID = q_srv.ID;
      pre_q_srv.Per_Busy = old_per_busy;
      pre_q_srv.Q_Stats.Per_Full = old_per_full;
      pre_q_srv.Q_Stats.Num_In_Q = old_in_q;
      pre_q_srv.Q_Stats.Num_Through_Q = old_through_q;

      *(ptr->LP_Stats[q_srv.ID].Q_Srv_Stats) = pre_q_srv;
  }
}
srv_oldnum[ID] = srv_num[ID];
 old_per_busy = q_srv.Per_Busy;
 old_per_full = q_srv.Q_Stats.Per_Full;
```

```
        old_in_q = q_srv.Q_Stats.Num_In_Q;
        old_through_q = q_srv.Q_Stats.Num_Through_Q;

        Complete = Is_Complete(Param,head);
      fprintf(stderr,"0ollector, complete = %d",Complete);

      if (Complete && head->Interval_num == final_num) {
         Done = Is_Done(Param,head);
         if (!Done) {
              Complete = FALSE;
         }
      }
   }

or
     (Complete && !Done && !Term):
      fprintf(stderr,"0ollector, complete");
        Complete = FALSE;
        Send_Stats(Param,head);

      if (final_num != 0) {
          ptr = Get_Ptr(Param,head,final_num);
            Complete = Is_Complete(Param,ptr);
            if (Complete)
               Done = Is_Done(Param,head->Next);
      }
        Send_File(Param,head,final_num);
        num = num++;
        Free_Record(Param,&head);
        if (!Done)
          Continue = Wait_Response(Param);
         printf(stderr,"0ollector, continue = %d",Continue);

or
     (Complete && Done && !Term):
      fprintf(stderr,"0ollector, Done");
        Send_Stats(Param,head);
      Send_File(Param,head,final_num);
        Param->All_LP.Term_Pid.term( );
        accept term( )
          { Term = TRUE; }

or
   (!Continue && !Done && !Term):
      fprintf(stderr,"0ollector, Ready to term");

        for(;;)
          select {
           accept src_stats(src_rec) { };
         or  accept sink_stats(sink_rec) { };
         or  accept srv_stats(q_srv_rec) { };
```

```
            or  accept term( ) {
                    Term = TRUE;
                     break;
            };
          }

    or
        (Term):
             terminate;
    }
}
```

```
#include "define.h"

/*    **************************************************************    */
/*    Process Body Terminate( )                                         */
/*    **************************************************************    */
/*                                                                      */
/*    Once this Terminate process accepts a signal from the             */
/*    Collector process, it submits transaction calls to all            */
/*    processes of the simulator to be ready to terminate.              */
/*                              .                                       */
/*    **************************************************************    */


process body Terminate( )
{
     register                  i;                    /*  Index                       */
     struct                    Term_Param Param;     /*  Initial paramter table      */

                                                     /*  Accept initial paramters    */
     accept setup(term_param)  {Param = term_param; };

  for(;;) {
    select {
      accept term( ) { };
        fprintf(stderr,"0erm, accept term");

        for (i = 0; i < Param.All_LP.Total_LP; i++) {
          switch(Param.All_LP.Type[i])
          {
            case SOURCE:
              fprintf(stderr,"0erm, call source");
              Param.All_LP.LP_Pid[i].Src_Pid.term( );
              break;

            case BRANCH:
              fprintf(stderr,"0erm, call branch");
              Param.All_LP.LP_Pid[i].Branch_Pid.term( );
               break;

            case QUE_SRV:
              fprintf(stderr,"0erm, call queue[%d]",i);
               Param.All_LP.Q_Pid[i].term(  );
              fprintf(stderr,"0n Term, call server[%d]",i);
              Param.All_LP.LP_Pid[i].Srv_Pid.term( );
               break;
            case SINK:
              fprintf(stderr,"0erm, call sink");
              Param.All_LP.LP_Pid[i].Sink_Pid.term( );
               break;
           }
        }

        fprintf(stderr,"0erm, call collector");
        Param.All_LP.Col_Pid.term( );

       or terminate;
    }
  }
}
```

Appendix B : One Example of an Input Model and Its Final Statistical Report

B.1: Format of the Input Model

In order to show an example of the input model and its statistical result, their formats have to be discussed first. The formats are reprinted from Vopata's thesis[VOPA89] under his permission. Figure B.1 shows the BNF notation for the input model and Figure B.2 gives a description of the BNF nonterminals.

```
<Start>      ::= <Begs> [ <Node> ]* <Ends> <SoS>
<Begs>       ::= ( ( 99 0 ) )
<Ends>       ::= ( ( 99 1 ) )
<SoS>        ::= ( ( 98 0 ) <Term_Time>  <Interval> )
<Node>       ::= <Source> | <Sink> | <Q_Server> | <Branch>
<Source>     ::= ( ( 0 ID ) ( <Stoch> ) <Mach> <Virt> <Gen> <Out_ID> )
<Sink>       ::= ( ( 1 ID ) <Mach> <Virt> <Num_In> )
<Q_Server>   ::= ( ( 2 ID ) ( <Stoch> ) <Mach> <Virt> <Out_ID>
                 <Q_Size> <Q_Method> <Num_In> )
<Branch>     ::= ( ( 3 ID ) <Mach> <Virt> <Num_In> <Num_Out>
                 ( <Out_list> ) )
<Out_list>   ::= [ ( Out_ID Prob ) ]1-5
<Stoch>      ::= ( <Type> <Min> <Max> <Arg1> [ <Arg2> ] )
<Out_ID>     ::= <ID>
```

Figure B.1: BNF Notation for the Input Model

<ID>	:: an unique number for each logical process
<Mach>	:: an unique number for each minicomputer (a list of these values is given in Figure B.3)
<Virt>	:: a Virtual Processor number (not used)
<Gen>	:: the number of message a source logical process is allowed to generate. If <Gen> = 0 then the source is allowed to generate an infinite number of messages.
<Num_In>	:: the number of incoming lines to a logical process
<Num_Out>	:: the process id of the destination logical process
<Q_Size>	:: the size of the queue buffer, must be greater than zero
<Q_Method>	:: the method for dequeueing message from the queue buffer <Q_Method> = 0 is FIFO <Q_Method> = 1 is LIFO <Q_Method> = 2 is SIRO <Q_Method> = 3 is PRIO
<Prob>	:: the probability of selecting the outgoing line The sum of all the probabilities of an "Out_Line" must total one (1).

<Type>	:: the type of stochastic distribution function
<Min>	:: Minimum cutoff for the distribution function If <Min> = 0 then Min is ignored
<Max>	:: Maximum cutoff for the distribution function If <Max> = 0 then Max is ignored
<Arg1>	:: First argument for the distribution function
<Arg2>	:: Second argument for the distribution function
<Term_Time>	:: Termination Time specified by the graphics front-end
<Interval>	:: Time Intervals (of simulated time) for sending collective status reports

Figure B.2: Description of the BNF Non-Terminals in Figure B.1

In the BNF notation, the "[<X>]" indicates that <X> is optional, the "[<X>]*" indicates that zero or more occurrences of <X>, and the "[<X>]1-5" indicates that there may be one to five occurrences of <X>.

Machine Number	Machine Host Name	Model
0	foxtrot	3B2/400
1	golf	3B2/400
2	hotel	3B2/400
3	india	3B2/400
4	juliet	3B2/400
5	kilo	3B2/400
6	lima	3B2/400
7	mike	3B2/400
8	november	3B2/400
9	hack	3B2/400
10	alpha	3B2/310
11	bravo	3B2/310

12	charlie	3B2/310
13	delta	3B2/310
14	echo	3B2/310
15	phobos	3B15
16	deimos	3B15

Figure B.3: List of Machines used in Figure B.2

Figure B.4 shows an example of the input model of the queueing network model.

```
( ( 99 0 ) )
( ( 0 0 ) ( 2 0 0 0.2 ) 8 0 0 1 )
( ( 3 1 ) 8 0 1 2 ( ( 2 0.60 ) ( 3 0.40 ) )
( ( 2 2 ) ( 2 0 0 0.1 ) 8 0 4 10 0 1 )
( ( 2 3 ) ( 2 0 0 0.1 ) 8 0 4 10 0 1 )
( ( 1 4 ) 8 0 2 )
( ( 99 1 ) )
( ( 98 0 ) 10 2)
```

Figure B.4: Example of an Input Model

B.2: Format of the Statistical Report

Figure B.5 shows the BNF of the statistics report and Figure B.6 gives a description of the BNF nonterminals[VOPA89].

```
<Start>       ::= [ <Message> ] <Interval> [ <Node> ]* "($$)"
<Message>     ::= "(end)" | "(abort)" | "(deadlock)"
<Node>        ::= <Source> | <Q_Server> | <Sink>
<Source>      ::= ( <ID>< Num_Left> )
<Q_Server>    ::= ( <ID> <Per_Busy> <Per_Full> <In_Q> <Through_Q> )
<Sink>        ::= ( <ID> <Num_Sunk> )
```

Figure B.5: BNF Notation for the Collective Report

(end) :: indicates that the following report is the last report

(abort)	:: indicates that an error occurred and the simulator is aborting the simulation
(deadlock)	:: indicates that model deadlock has occurred (not used)
($$)	:: indicates the end of the current statistics report
<Interval>	:: simulation time of the current statistics report
<ID>	:: the unique identifier of each logical process
<Num_Left>	:: the number of remaining messages that a source process has left. If the source was to generate an infinite number of messages, the value will be negative and will represent the number of message that the source has generated.
<Per_Busy>	:: the percent utilization of a server process
<Per_Full>	:: the percent capacity of a queue process
<In_Q>	:: the number of messages currently in a queue process
<Through_Q>	:: the number of messages that have passed through a queue process
<Num_Sunk>	:: the number of messages discarded by a sink process

Figure B.6: Description of the BNF Non-Terminals in Figure B.5

Figure B.7 shows the final statistics report of the queueing network model described in Figure B.4.

```
(10.0000)
(0 -10)
(2 50.0000 70.0000 7 3)
(3 40.0000 70.0000 7 3)
(4 2)
($$)
```

Figure B.7: Example of a Statistics Report

Deadlock Avoidance in a Distributed Simulation System

by

Li-Fang L. Hsieh

B.A. National Taiwan University, 1977

AN ABSTRACT OF A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computing and Information Sciences

KANSAS STATE UNIVERSITY

Manhattan, Kansas

1989

Abstract

This project uses a concurrent language, Concurrent C [GEHA88], to implement a distributed discrete-event simulator. It runs under a deadlock avoidance algorithm proposed by Chandy and Misra [CHAN79] and it adopts a basic queueing network scheme [SAUE80] from an RESQ simulation package. A user can send input data either from a file or from a stream socket to initiate the distributed simulator. Based on the user's specification, the simulator will execute the model either in a centralized or distributed mode.
When the interval time for a statistical report expires, the simulator will collect all the necessary information and send it back to the user. The project thus explores the interesting area of using a concurrent language to implement a distributed simulator in a parallel execution environment.