Distributed Discrete-Event Simulation in Concurrent C

by

Edward William Vopata

B. S., Kansas State University, 1986

A MASTER'S THESIS

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:

Virgil E. Wallentine - Major Professor

LD
2668
.T4
CMSC
1989
V67
c. 2

A11208 301223

Table of Contents

1. Introduction 2

2. Simulation 5

2.1. Concepts of Simulation 5

2.2. Causality 6

2.3. Queueing Networks 7

2.4. Traditional Simulation 9

2.5. Problems with Traditional Simulation 10

3. Distributed Simulation 11

3.1. Waiting Rules for Logical Processes 13

3.2. Deadlock 15

3.3. Distributed Simulation Strategies 18

4. Implementation of the Distributed Simulator 23

4.1. Motivation for Implementation 23

4.2. Programming Languages 23

4.3. The Distributed Simulation Environment 25

4.4. The Distributed Discrete-Event Simulator 26

4.5. Deadlock Detection and Resolution 37

5. Summary and Future Research 48

References 51

Appendix A: Multiprocessor Computer Systems 53

Appendix B: Message Passing 56

Appendix C: Graphics Simulator Interface 58

Appendix D: Stochastic Distribution Functions 64

Appendix E: Implementation Notes 66

Appendix F: Concurrent C Source Code for the Distributed Simulator 71

List of Figures and Tables

Figure 1.1
- The Distributed Simulation Environment 2

Figure 2.1
- Model of a Single Processor Computer System 5

Figure 2.2
- State Transitions [FRAN77] 7

Figure 2.3
- Symbols of a Queueing Network Modeling 8

Figure 2.4
- Model of a Queueing Network 9

Figure 3.1
- Simulation Model 12

Table 3.1
- Operation of Simulation Model in Figure 3.1 13

Figure 3.2
- Simulation Model using the Waiting Rules 14

Table 3.2
- Operation of Simulation Model in Figure 3.2 15

Figure 3.3
- Deadlock Model 16

Table 3.3
- Operation of the Deadlock Model in Figure 3.3 16

Figure 3.4
- Waiting Conditions for the Deadlocked Model in Figure 3.3 17

Figure 3.5
- Gridlock: An Example of Model Deadlock 18

Figure 3.6
- Queueing Network Model of the Gridlock Example 18

Figure 3.7
- Example of Deadlock Avoidance 19

Figure 4.1
- Distributed Simulation Environment 25

List of Figures and Tables (cont.)

Figure 4.2
- Concurrent C Code for Creating Logical Processes 27

Figure 4.3
- Process Layout of a Distributed Simulation Model 27

Figure 4.4
- Concurrent C Code for Receiving Incoming Messages 31

Figure 4.5
- General Algorithm for the Source Process 32

Figure 4.6
- General Algorithm for the Queue Process 33

Figure 4.7
- General Algorithm for the Server Process 35

Figure 4.8
- General Algorithm for the Sink Process 35

Figure 4.9
- General Algorithm for the Branch Process 36

Figure 4.10
- Probability of Selecting an Outgoing Line 37

Figure 4.11
- General Algorithm for Selecting an Outgoing Line 37

Figure 4.12
- An Example of Local and Global Deadlock and Local and Global Deadlock Detection 39

Figure 4.13
- General Algorithm for the Deadlock Resolver Process 40

Figure 4.14
- Concurrent C Code for Receiving Send and Res_Send Messages 43

Figure 4.15
- Deadlock Resolution using intercept_transaction() 44

Figure 4.16
- Deadlock Resolution using the Get_Time transaction 45

List of Figures and Tables (cont.)

Figure A.1
- A Tightly Coupled Multiprocessor Computer System 53

Figure A.2
- A Loosely Coupled Multiprocessor Computer System 54

Figure A.3
- The Distributed Simulation Environment on the KSU.CIS Network of Minicomputers 55

Figure C.1
- BNF notation for the Input Model 59

Figure C.2
- Description of the BNF Non-Terminals in Figure C.1 59

Figure C.3
- List of the Machines unsed in Figure C.2 60

Figure C.4
- Example of an Input Model 61

Figure C.5
- BNF Notation for the Collective Report 61

Figure C.6
- Description of the BNF Non-Terminals in Figure C.5 61

Figure C.7
- Example of a Collective Statistics Report 62

Figure C.8
- Simulation Control Messages 63

Figure D.1
- Stochastic Distribution Functions 64

Figure E.1
- Command Line Switches 66

Figure E.2
- Stats_Report 67

Figure E.3
- Size_of Report 68

Figure E.4
- Out_Graph Report 70

1. Introduction

In recent years multiprocessor computer systems, such as networks of mini- and microcomputers, have become an attractive alternative to traditional uniprocessor computers. The programs that run on these multiprocessor computer systems are called distributed programs and are composed of multiple interacting processes. Each process runs on a separate processor and uses message passing to interact with other processes. Distributed programming languages, such as Ada [STAM85] and Concurrent C [GEHA88], can be used to develop these distributed programs. The programming languages provide many language level facilities that make distributed programming easier.

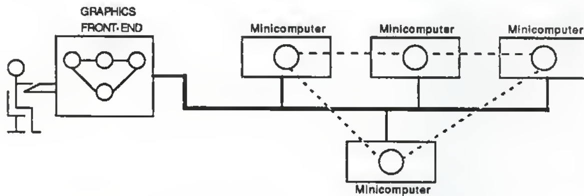


Figure 1.1: The Distributed Simulation Environment

There are many applications for distributed programming. One such application is distributed simulation [MISR86], [REED87]. This thesis describes the implementation of a distributed discrete-event simulator written in Concurrent C. The distributed simulator is part of a distributed simulation environment for the development and simulation of queueing network models. The distributed simulation environment runs on a multiprocessor computer system and is composed of the distributed simulator and a graphics front-end (Figure 1.1). The graphics front-end allows users to graphically create queueing

network models. These models are then sent to the distributed simulator where they are simulated.

The simulation of queueing network models has long been a task where the computational requirements needed to simulate large and complicated models have far exceeded the computational capabilities of the fastest available machines [REED87]. The traditional approach to simulation is to create a simulator that runs on a uniprocessor computer. Simulations conducted on a uniprocessor simulator are generally slow. Distributed simulation offers an alternate approach by distributing the simulation on a multiprocessor computer system.

There are several problems involved in the development of a distributed discrete-event simulator. The major problem is that the processes in the distribed simulation program may deadlock. There are several strategies for coping with the deadlock problems [JEFF85], [REED87]. The distributed simulator discussed in this thesis uses a deadlock detection and resolution strategy [CHAN81]. The deadlock detection and resolution strategy is composed of a deadlock detection mechanism for detecting and reporting occurrence of deadlock, and a deadlock resolution mechanism for resolving the reported deadlocks. The deadlock detection mechanism was implemented by Scott Hammond [HAMM88] as part of the kernel of Concurrent C and the deadlock resolution mechanism was implemented as part of the distributed simulator. One of the goals of this thesis is to determine the practicality of having a deadlock detection mechanism at the language level and a deadlock resolution mechanism at the application level.

A distributed discrete-event simulator was implemented in the distributed programming language Concurrent C and is capable of

simulating a wide variety of queueing network models that are sent by the graphics front-end. These queueing network models are simulated on a network of minicomputers. The language level deadlock detection mechanism detects and reports deadlock, and the application level deadlock resolution mechanism resolves the deadlock. The distributed simulator also allows the graphics front-end to monitor the progress of a simulation by sending statistical reports about the current state of the simulation to the graphics front-end.

Chapter 2 discusses the basic concepts of simulating queueing networks and the problems with the uniprocessor approach to simulation. The basic concepts and problems of distributed simulation are discussed in Chapter 3. Chapter 3 also surveys several distributed simulation strategies for coping with the problems of deadlock. Chapter 4 describes the implementation of the distributed simulator in Concurrent C and the deadlock detection and resolution strategy used by the simulator. Finally, Chapter 5 discusses the conclusions and future research projects.

2. Simulation

2.1. Concepts of Simulation

Simulation is the creation and execution of a model of a physical system. A system is an organized collection of independent interacting elements that functions as a single unit. A physical system is a real world system [REED87]. The simulation model is an abstract representation, such as a computer program or a diagram. Figure 2.1 shows a model of a physical system.

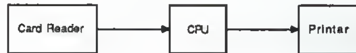


Figure 2.1: Model of a Single Processor Computer System

A physical system can be conceptualized as a state machine. The current state of the physical system is described by a system state, and its behavior over time is described by a sequence of state transitions. In the simulation model, the state of the system is modeled by a collection of state variables, and state transitions are modeled by events. The occurrence of an event causes the state variables to be modified. The set of values assigned to the state variables after an event has occurred represents the current state of the simulation, referred to as the simulation state. Each simulation state corresponds to a state in the physical system. Time in the physical system, referred to as physical time, is replaced in the simulation model by the notion of simulated time. Each event has a timestamp associated with it indicating the physical time when the corresponding

state transition occurs. [REED87].

In the simulation model, simulated time is maintained by a variable called the clock or simulation clock [MISR86]. The clock holds the current value of simulated time, which is the time up to which the physical system has been simulated. There are two methods for advancing the simulation clock. In time-driven simulation the clock is advanced one "tick" at a time, and every event that occurs at that time is simulated. In event-driven simulation the clock is advanced to the time of the next event, and then that event is simulated. Only event-driven simulation will be discussed in this thesis.

2.2. Causality

One important aspect of simulation is maintaining causality. With the exception of time travel in science fiction novels, physical systems always obey the causality principle. This simply states that the future cannot affect the past. Causality imposes a time ordering on state transitions in the physical system. If a state transition has some effect on another state transition the former must always occur before the latter, i.e., the cause must always precede the effect. [REED87].

The time ordering of state transitions in the physical system imposes a time ordering on events in the simulation. In particular, the order that events are simulated must be consistent with the order that state transitions occur if the simulation is to faithfully reproduce the behavior of the physical system. For example, if state transition A in the physical system occurs at time 3 and has some influence on transition B occurring at time 5, then the simulation must

model event A before modeling event B. If event B is simulated before event A, the simulation would not accurately model the physical system. When events are not simulated in the correct sequence, causality is violated and the simulation is said to be incorrect. [REED87].

Physical systems are classified by the manner in which state transitions occur [FRAN77]. In continuous-time systems, state transitions occur gradually over a period of time (Figure 2.2 (a)). Weather is an example of a continuous-time system. In discrete-time systems, state transitions occur at specific instances in time (Figure 2.2 (b)). Discrete-time systems are often referred to as discrete-event systems because the specific instances correspond to occurrences of events. Queueing networks are examples of discrete-event systems. This thesis will only consider the simulation of discrete-event systems and specifically, the simulation of queueing networks.

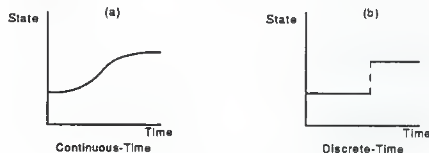


Figure 2.2: State Transitions [FRAN77]

2.3. Queueing Networks

A queueing network is a collection of stations (or service centers) arranged in such a way that customers proceed from one to another in order to fulfill their service requirements. In queueing networks of computer systems, the stations represent the various system resources (e.g. CPU's, channels, disk, and drums) while the

customers correspond to jobs in the system [BRUE80].

Each station has an associated queue in which jobs may wait prior to receiving service. The stations are characterized by a service time and the queues are characterized by a queueing discipline. The service time is the amount of time required to service a job. The queueing discipline determines the order in which jobs are removed from the queue. "First In, First Out" (FIFO) and "Last In, First Out" (LIFO) are two examples of queueing disciplines [BRUE80].

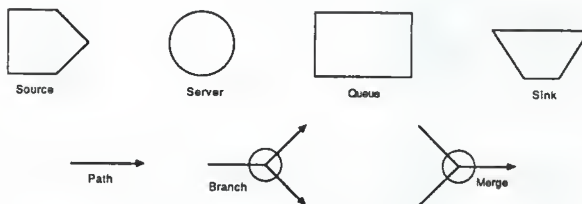


Figure 2.3: Symbols of a Queueing Network Model

Figure 2.3 shows several of the common symbols used to describe queueing networks [GHOS85]. These symbols represent stations, queues, and important points in the physical queueing network system. A server symbol represents a station and a queue symbol represents a queue. A source symbol represents the point where jobs enter the system. A sink symbol represents the point where jobs leave the system. The flow of jobs through the system is represented by a path. Branch symbols show points where jobs may follow one of several possible paths. Merge symbols show points where jobs on multiple paths merge onto a single path. Figure 2.4 shows a model of a queue network in terms of these symbols.

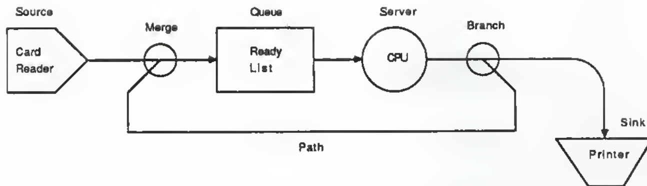


Figure 2.4: Model of a Queueing Network

2.4. Traditional Simulation

The traditional approach to simulation is to create and execute a simulation program on a uniprocessor computer. The simulation program or simulator is composed of procedures. These procedures simulate the behavior of the various elements in the physical system. For queueing networks these procedures simulate the behavior of sources, servers, queues, sinks, branches, and merges. Generally, there is only one procedure for each type of element. Events model the arrival and departure of jobs at a particular element. A procedure simulates the behavior of an element by updating various state variables and by scheduling new events as necessary.

Causality in the uniprocessor simulator is maintained by means of a global clock variable and a data structure called the event list. This event list contains a list of scheduled events. Simulation progresses by removing the oldest event from the event list, i.e. the event with the smallest timestamp. The clock is then advanced to the time of the event and a procedure is called to simulate the event. After the procedure has simulated the event, the simulation is ready

to process the next event. By always processing the oldest event first causality is never violated.

2.5. Problems with Traditional Simulation

There are several problems with the traditional approach to simulation. The major problem is that uniprocessor simulations are often slow. The single processor must both schedule and simulate every event in the model, and as the number of events increases so does the load on the single processor. Another problem is that uniprocessor computers have a limited amount of physical memory space. The limited memory space restricts the size of the event list and the number of events that can be scheduled at any given instance in simulated time.

3. Distributed Simulation

An alternative to the traditional approach to simulation is distributed simulation. Distributed simulation is the creation and execution of a simulation program on a multiprocessor computer system. A multiprocessor computer system consists of multiple processors connected by a common bus or network. Multiprocessor computer systems are discussed in Appendix A. The programs that run on multiprocessor computer systems are composed of multiple interacting program routines called processes. These processes are executed concurrently on separate processors and interact by exchanging messages. Process interaction by message passing is discussed in Appendix B.

A physical system can be visualized as some number of independent, concurrently executing elements, i.e. physical processes, that interact in some fashion. A natural method for partitioning a distributed simulation program is to create a simulation model that is topologically identical to the physical system. Each physical process is modeled by a separate simulation routine called a logical process (LP). Interactions between physical processes are modeled by exchanging timestamped messages between corresponding logical processes. The timestamp denotes the simulation time when the event occurs in the receiving process. [REED87]. Partitioning a queueing network model involves the creation logical processes for each source, server, queue, branch, and merge in the simulation model.

A few basic rules must be obeyed by each logical process. A logical process can receive and read the contents of messages, can generate and send new messages, and can update its internal state variables. No variables are shared between distinct logical processes.

Each process maintains a local clock variable containing the current value of simulated time, i.e. the timestamp of the last message processed by the logical process. The timestamp on each message generated by a process must be at least as large as the local clock of the process sending the message. Otherwise, processes could create events "in the past," a clear violation of causality. [REED87].

The execution of a distributed simulation is illustrated in Figure 3.1 and Table 3.1. Figure 3.1 shows a distributed simulation model and Table 3.1 shows the events that occur during the execution of the model. The simulated time in Table 3.1 shows the time at which each event occurs. The first event that occurs is the servers (LP2 and LP4) requesting the next message from their respective queues (LP1 and LP3). Since both queues are empty, the servers must wait until their requests are granted. At time 3 the source (LP0) generates a simulated job and sends the message (M1,3) to LP1. The message is a tuple representing the simulated job (M1) and a the time of the event (3). The queue (LP1) receives the message (M1,3) and accepts LP2's request. LP2 receives the message (M1,3) and simulates servicing the simulated job.

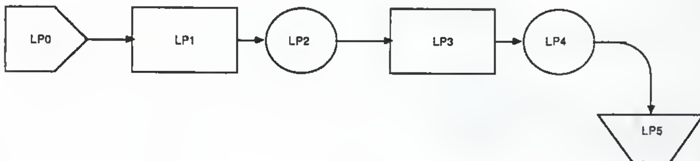


Figure 3.1: Simulation Model

Table 3.1: Operation of Simulation Model in Figure 3.1

Simulated Time	Source LP0	Queue LP1	Server LP2	Queue LP3	Server LP4	Sink LP5
0			Request Next msg		Request Next msg	
3	Send(M1,3)	Recv(M1,3) Accept req.	Recv(M1,3)			
5	Send(M2,5)	Recv(M2,5)				
13		Accept req.	Send(M1,13) Request Next msg Recv(M2,5)	Recv(M1,13) Accept req.	Recv(M1,13)	
23			Send(M2,23) Request Next msg	Recv(M2,23) Accept req.	Send(M1,23) Request Next msg Recv(M2,23)	Recv(M1,23)
33					Send(M2,33) Request Next msg	Recv(M2,33)

A server services a simulated job by updating the timestamp of the message and sending the updated message to the next logical process. The formula for updating the timestamp of a messages is dependent upon the timestamp of the (current) message (CTS), the timestamp of the last message sent (LTS), and the service time (ST) of the server. The formula for updating the timestamp of the current message is: Maximum(CTS,LTS) + ST. If LTS is larger than CTS, the server was busy processing the last simulated job when the current message arrived and the current message had to wait in the queue for LTS - CTS time units. If, however, CTS is larger than LTS, the server was idle for CTS - LTS time units.

3.1. Waiting Rules for Logical Processes

An important concern in distributed simulation is insuring that the principles of causality described in Chapter 2.2 are not violated. Causality, in traditional uniprocessor simulation, was easily maintained by only processing one event (message) at a time, but in

distributed simulation, multiple events (messages) can be processed simultaneously, as shown in Table 3.1 at simulated time 23. The distributed simulation program must, therefore, prevent causality from being violated. One method for preventing causality violations is to apply the "Waiting Rules" for logical processes described in [CHAN81]. These rules specify how logical processes may send and receive messages. The Waiting Rules for logical processes are as follows:

(1) A logical process must wait on every possible outgoing line on which it has sent a message until that message is received.

(2) A logical process must wait until a message has arrived on every possible incoming line before selecting the message with the smallest timestamp to receive.

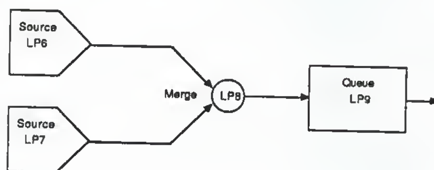


Figure 3.2: Simulation Model using the Waiting Rules

Figure 3.2 and Table 3.2 illustrates the execution of a simulation model with respect to the Waiting Rules. The source (LP6) sends the message (M3,8) to the merge (LP8). The merge does not have a message on every possible incoming line so LP6 must wait until LP8 receives the message. When the source (LP7) sends the message (M4,10), LP8 then has a message on every possible incoming line and can select the message with the smallest timestamp (M3,8). LP8 receives the message (M3,8) and frees LP6. LP8 sends the message (M3,8) to the queue LP9, where it is received. LP6 then sends its next message (M5,10) to LP8. The merge now has a message on every possible incoming line, but both message have the same timestamp.

Therefore, LP8 selects one of the messages and sends it on. The other message remains on the line until a message with a larger timestamp arrives.

Table 3.2: Operation of Simulation Model in Figure 3.2

Source LP6	Source LP7	Merge LP8	Queue LP9
Send(M3,8)		- -	
	Send(M4,10)	Recv(M3,8) Send(M3,8)	Recv(M3,8)
Send(M5,10)		Recv(M5,10) Send(M5,10)	Recv(M5,10)
Send(M6,11)		Recv(M4,10) Send(M4,10)	Recv(M4,10)

3.2. Deadlock

The major problem with the application of the waiting rules is that they can cause deadlock [REED87], [CHAN81]. Deadlock is a state where a set of logical processes is waiting for an event that will never occur. Figure 3.3 and Table 3.3 show the execution of a simulation model that deadlocks. The branch (LP11) in the model has a high probability of sending message to the queue (LP12) and the queues (LP12 and LP14) can only hold two messages.

Deadlock occurs when all the message received by the branch (LP11) are sent to the queue (LP12). The first message received by LP12 is given to the server (LP13). LP13 processes the message, updates the timestamp, and tries to send the message (M7,25) to the merge (LP16). LP16 has two incoming lines and must wait for a message to arrive from LP15. Therefore, LP13 must wait until LP16 receives the message. In the mean time, LP10 and LP11 have filled the queue (LP12) and LP12 is no longer receiving messages. LP10 then sends another message to LP11. LP11 receives the message and tries to send

it to LP12, but cannot, since the queue is full. Therefore LP11 must wait until LP12 receives the message. LP10 then tries to send another message to LP11, but cannot, since LP11 is not receiving. LP16 is still waiting for a message from the server (LP15), LP15 is waiting for the queue (LP14) to grant its request for the next message, and the queue (LP14) is empty and is waiting for a message to arrive from LP11. Since LP11 is still trying to send a message to LP12, the event of LP11 sending a message to LP14 will never occur and a deadlock state exists.

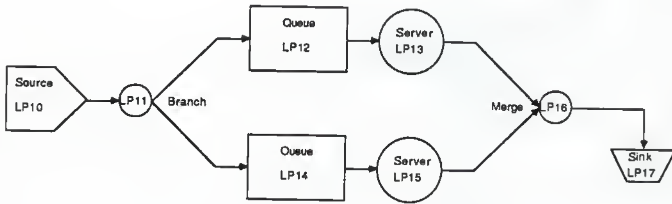


Figure 3.3: Deadlock Model

Table 3.3: Operation of the Deadlock Model in Figure 3.3

Source LP10	Branch :LP11	Queue LP12	Server LP13	Queue LP14	Server LP15	Merge LP16	Sink LP17
			Request Next message		Request Next message		
Send(M7,15)	Recv(M7,15) Send(M7,15)	Recv(M7,15)	Recv(M7,15) Send(M7,25) <waiting>			..	
Send(M6,16)	Recv(M6,16) Send(M6,16)	Recv(M6,16)					
Send(M9,20)	Recv(M9,20) Send(M9,20)	Recv(M9,20) <Queue FULL>					
Send(M11,26)	Recv(M11,26) Send(M11,26) <waiting>						
Send(M10,25) <waiting>							
waiting for LP11 to Recv message	waiting for LP12 to Recv message	Full Queue. Not Recv'ing waiting for LP13 to req. next message	waiting for LP16 to Recv message	waiting for LP11 to send a message	waiting for LP14 to grant request	has message from LP13, waiting for message from LP15	waiting for a message from LP16

Figure 3.4 shows the deadlocked simulation model. The arrows in the figure represent the direction in which the logical processes are waiting. The branch is said to be the critical point in the deadlock state because it is where the deadlock originates.

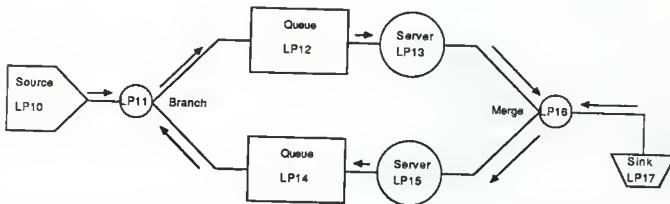


Figure 3.4: Waiting Conditions for the Deadlocked Model in Figure 3.3

The deadlock illustrated in Figure 3.3 and Table 3.3 is called "simulation deadlock" because it is caused by the waiting rules that are used to maintain causality. Another type of deadlock is called "model deadlock." Model deadlock is caused by the simulation faithfully modeling a physical system that deadlocks. Figure 3.5 shows a model of a physical system that deadlocks. This model depicts traffic deadlock or gridlock [PETE83]. Gridlock occurs when the streets or the grid are filled with vehicles and a vehicle blocks each intersection. Figure 3.6 shows a queueing network model of the gridlock system. The queues represent the streets in the system, the sources and sinks represent the flow of traffic into and out of the system, and the combined branch and merge points represent the intersections. Deadlock occurs when the queues are filled with messages and a message is waiting at each intersection.

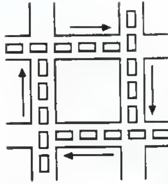


Figure 3.5: Gridlock: An Example of Model Deadlock

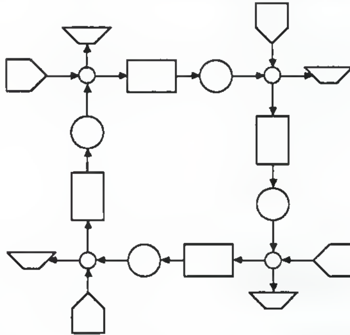


Figure 3.6: Queueing Network Model of the Gridlock Example

3.3. Distributed Simulation Strategies

There are three basic distributed simulation strategies for coping with the problem of simulation deadlock.

3.3.1. Deadlock Avoidance

The first distributed simulation strategy for coping with simulation deadlock is called deadlock avoidance [CHAN78]. The deadlock avoidance strategy uses the Waiting Rules explained in Chapter 3.1. Deadlock avoidance prevents deadlock from occurring by using NULL messages. A NULL message, in the form (NULL,timestamp), is sent from one

logical process to another to represent that the corresponding physical process did not send a message at that time. NULL messages are treated in much the same way as normal messages, except NULL message do not affect the state variables of a logical process other than the local clock.

Deadlock avoidance is handled by the branch LP's. Whenever a branch sends a normal message at time t (M,t) on one of its outgoing lines, the branch also sends a NULL message with the same timestamp (NULL,t) on all other outgoing lines. The NULL messages allow the logical processes to advance their local clock to the time of the timestamp, without fear of violating causality. The logical processes that receive the NULL messages, update the NULL messages and send them to the next logical process in the simulation.

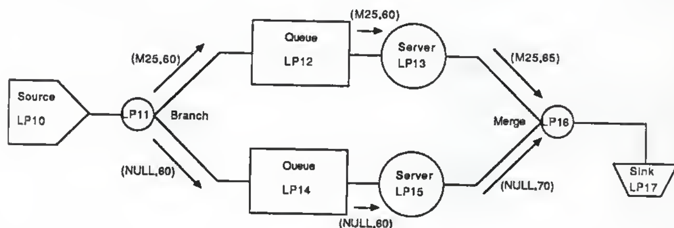


Figure 3.7: Example of Deadlock Avoidance

Figure 3.7 illustrates how the deadlock avoidance strategy can avoid the deadlock that occurs in Figure 3.4. The branch (LP11) sends the normal message (M25,60) along the upper path and the NULL message (NULL,60) along the lower path. The normal message (M25,60) propagates through the queue (LP12) to the server (LP13) where it is processed. LP13 then sends the normal message (M25,65) to the merge

(LP16). In the mean time, the NULL message (NULL,60) propagates through the queue (LP14) to server (LP15) where the NULL message is processed. LP15 then sends the updated NULL message (NULL,70) to the merge. The merge now has a message on every possible incoming line and can now select the message with the smallest timestamp, namely (M25,65).

The major disadvantage of the deadlock avoidance approach is that a large number of unnecessary NULL messages may be generated and sent. In addition to placing an extra burden on the communication system, a significant amount of processor time may be spent processing the NULL messages. Another problem is that the deadlock avoidance approach does not provide a mechanism for detecting model deadlock.

3.3.2. Time Warp

The second strategy for coping with simulation deadlock is called Time Warp or Virtual Time [JEFF85]. The Time Warp strategy copes with deadlock by disregarding the Waiting Rules described in Chapter 3.1 and allowing messages to arrive in any order, even at the risk of violating causality. The Time Warp strategy maintains causality by providing a rollback and recovery mechanism. Whenever a message arrives at a logical process that is not in the proper order (i.e. violates causality) the rollback and recovery mechanism rolls the simulation back to a point in simulated time (i.e. time warps the simulation back in time) before the error occurred (the rollback) and restarts the simulation at that point (the recovery). The rollback and recovery mechanism is complicated by the fact that the logical process may have sent messages before the erroneous message arrived. Therefore, the rollback and recovery mechanism must also provide the

means for causing other logical processes to rollback and recover.

The major disadvantages with the Time Warp approach are the overhead involved in saving the state information and performing the rollback and recovery, and the potentially large amount of memory required to implement it. The Time Warp approach also does not provide a mechanism for detecting model deadlock.

3.3.3. Deadlock Detection and Resolution

The third distributed simulation strategy for coping with deadlock, we will call deadlock detection and resolution. This strategy is also called the deadlock detection and recovery strategy [CHAN81]. The deadlock detection and resolution strategy, like the deadlock avoidance strategy, uses the Waiting Rules for logical processes (Chapter 3.1). The deadlock detection and resolution strategy involves running the simulation until deadlock occurs, detecting the deadlock, and resolving the deadlock, which will allow the simulation to continue.

The deadlock detection and resolution strategy described in [CHAN81] uses a special process called the "controller" to detect deadlock. Deadlock detection is based upon knowing the process states of the logical processes. The process state of a logical process determines whether the process is waiting for the arrival of a message, waiting for a message to be received, or neither. The controller examines the process states and looks for occurrences of "deadlocked knots" of processes, such as the example in Figure 3.4 where the deadlock knot of processes consists of LP11, LP12, LP13, LP14, LP15, and LP16.

Once deadlock has been detected, the controller initiates deadlock resolution. Deadlock resolution in the [CHAN81] strategy involves making a series of distributed computations that will allow the logical processes to advance their simulation clocks. The logical processes, especially the merge processes, are then allowed to temporarily disregard the Waiting Rules (Chapter 3.1) and accept all incoming messages up to their new simulated time. The description of the deadlock resolution computations appear in [CHAN81] and are summarized in [REED87].

4. Implementation of the Distributed Simulator

4.1. Motivation for Implementation

In this section we discuss the implementation of a distributed discrete-event simulator. The motivations for this thesis and the distributed simulator are as follows:

(1) to experiment with distributed programming and distributed programming languages

(2) to develop a distributed simulation environment for the simulation of queueing networks

(3) to determine the practicality of implementing a deadlock detection mechanism in the distributed kernel of a distributed programming language

(4) to develop a deadlock resolution mechanism for resolving the deadlock detected by the above deadlock detection mechanism

4.2. Programming Languages

The selection of a programming language in which to implement the distributed simulation is very important. The programming language must provide facilities for handling distributed process management and message passing, otherwise the programmer must create these facilities. The facilities provided by a programming language are generally easier to understand and much more reliable.

General purpose programming languages such as C, Fortran, and Pascal provide neither the distributed process management nor the message passing facilities needed by the distributed simulator. General purpose programming languages are, however, widely used in the implementation of traditional uniprocessor simulators.

Simulation programming languages such as Simula and Simscript are specifically designed for the implementation of traditional

uniprocessor simulators. Simulation programming languages, however, do not provide the required distributed process management nor the message passing facilities.

Concurrent programming languages such as Concurrent Pascal and Modula-2 provide a form of distributed process management. These languages are, however, concerned with the mutual exclusion of shared data objects and do not provide message passing facilities.

Distributed programming languages such as Ada and Concurrent C provide both the distributed process management and the message passing facilities required by the distributed simulator.

4.2.1. Reasons for Selecting Concurrent C

We decided to use Concurrent C over Ada because Concurrent C provides several facilities that Ada does not. First of all, Concurrent C provides both synchronous and asynchronous message passing facilities, in the form of *transactions* and *async transactions*. Ada only provides synchronous message passing facilities in the form of an *extended rendezvous*. Concurrent C provides facilities for altering the order in which messages are received based on the contents of the messages. Concurrent C does this by providing *suchthat* and *by* clauses in their *accept* statements. Ada's *accept* statements only allow messages to be received in FIFO order. The source code for the Concurrent C compiler and run-time libraries were available, but no source code for Ada was available at the time. The source code for the compiler and the run-time libraries was needed to aid in the implementation of the kernel level deadlock detection mechanism. Finally, Concurrent C was portable to our multiprocessor computer system.

4.3. The Distributed Simulation Environment

The distributed simulation environment is designed to provide the means for graphically creating and simulating models of queueing networks. The distributed simulation environment is composed of two interacting components: the graphics front-end and the distributed simulator (Figure 4.1). The distributed simulation environment runs on a multiprocessor computer system. The graphics front-end runs on a special purpose graphics machine and the distributed simulator runs on a homogeneous collection of minicomputers. The details of this multiprocessor computer system are described in Appendix A.

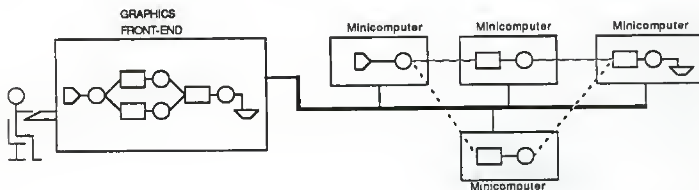


Figure 4.1: Distributed Simulation Environment

The graphics front-end provides the user interface for the distributed simulation environment. The graphics front-end allows the user to graphically create and edit queueing network models. These models will be sent to the distributed simulator where they will be simulated. The simulator allows the graphics front-end to monitor the progress of the simulation by providing statistical reports about the state of the simulation. The graphics front-end takes these reports and graphically displays them. A description of the graphics front-end is given in [BUTLER].

The distributed simulator provides the means for simulating the queueing network models. The distributed simulator take a simulation model (input model) sent from the graphics front-end and creates a simulation in accordance with the input model. The distributed simulation then executes the model on a multiprocessor computer system and collects statistical reports regarding the state of the simulation. These reports are sent to the graphics front-end. The distributed simulator is also responsible for coping with any deadlock states that may occur.

4.4. The Distributed Discrete-Event Simulator

4.4.1. Starting the Distributed Simulator

The distributed simulator is only started when the graphics front-end is ready to send an input model. A "connection program" (Appendix F, connect.c) opens a connection to the graphics front-end and starts the "main process" of the distributed simulator on a single processor. The main process receives the input model from the graphics front-end and creates the simulation.

The input model is a numeric representation of the graphical queueing network model displayed by the graphics front-end. The input model specifies the elements involved in the simulation and additional information regarding the operation of the elements. These elements will become logical processes in the simulation. Appendix C describes the format of the input model and the communication interface between the graphics front-end and the distributed simulator.

Once the main process has received the input model it proceeds to create the simulation. A logical process is created for each element

in the input model on the processor specified by the input model. Figure 4.2 show the Concurrent C code for the creation of each type of logical process on processor X. The process id (pid) returned by a create statement is used to allow other processes to send the created process messages.

```
Pid_src = create Source () processor (X);
Pid_srv = create Server () processor (X);
Pid_que = create Queue  () processor (X);
Pid_snk = create Sink   () processor (X);
Pid_brn = create Branch () processor (X);
```

Figure 4.2: Concurrent C Code for Creating Logical Processes

After all the logical processes have been created, the main process creates a collector process, a terminator process, a set of kernel level deadlock detection processes, and a set of deadlock resolver processes. The collector and the terminator processes are created on the same processor as the main process. A kernel level deadlock detector process and a deadlock resolver process are created on every processor involved in the simulation (See Figure 4.3).

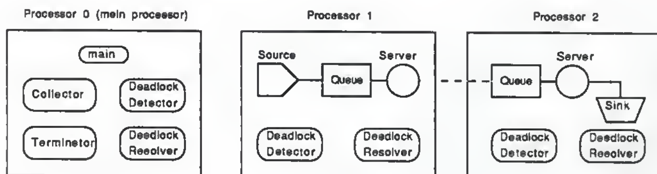


Figure 4.3: Process Layout of a Distributed Simulation Model

Once all the processes have been created, the main process sends a "setup message" to each process (P). This setup message contains a

list of process id's of the processes to which the process (P) can send messages, and any additional information required by the process. The additional information is provided by the input model.

4.4.2. Supporting the Distributed Simulator

The collector and the terminator are processes that support the operation of the distributed simulator. The collector is responsible for collecting and sending statistical reports to the graphics front-end and the terminator is responsible for the termination of the simulation.

Each logical process maintains a local simulation clock and a collection of state variables. These state variables describe the current state of the logical process. At predetermined intervals of simulated time as specified in the input model, each logical process will compose a "status report" of its current state variables and send the report to the collector. After the collector has received a status report from each logical process in the simulation, it sends a "collective report" to the graphics front-end. The format of this collective report is described in Appendix C.

After sending the collective report to the graphics front-end, the collector waits for the graphics front-end to reply with a control message. This control message is described in Appendix C and will tell the collector to either continue the simulation or initiate "simulation termination." The format of the control messages is described in Appendix C. The collector waits for a control message and does not accept status reports from the logical processes. The logical processes can still send status reports to the waiting collector,

but they must wait until the collector accepts the reports. This keeps the logical processes from flooding the collector with status reports and allows the graphics front-end to control the distributed simulation. The Concurrent C source code for the collector process is given in Appendix F (col.cc).

Three conditions can cause the collector to initiate simulation termination: 1) the collector receives a terminate control message from the graphics front-end, 2) the collector receives a status report from a logical process whose simulation time exceeds a predetermined threshold, or 3) the collector determines that all the messages generated by the source processes have been discarded. The predetermined simulation time threshold is provided by the input model. The input model can also specify the number of simulated jobs (messages) that a source can produce or specify that the source can produce an infinite number of jobs. In the latter case, the third condition is meaningless. After one of these termination conditions has been met the collector initiates simulation termination by sending a "terminate message" to the terminator process.

The terminator process, after receiving a terminate message from the collector, sends a terminate message to every logical process in the simulation. When a logical process receives a terminate message it enters a termination phase. The termination phase consists of sending a final status report to the collector, receiving and discarding any outstanding messages, and terminating. After all the terminate messages have been sent to the logical processes, the terminator sends a terminate message to the collector and terminates. After receiving the terminate message from the terminator, the collector

collects all the final statistical reports from the logical processes, sends a final collective report to the graphics front-end, and terminates. The Concurrent C source code for the terminator process is given in Appendix F (term.cc).

The distributed simulator also makes use of several stochastic distribution functions. These functions are used to generate random variables. Each function is based on a particular statistical distribution formula as described in [HALL88]. Stochastic distribution functions are used to generate random variables for arrival times, service time, and probabilities. Stochastic distribution functions are used by the source, server, and branch logical processes. The type of stochastic distribution function and the parameter for the function are specified in the input model. The stochastic distribution functions used by the distributed simulator are discussed in Appendix D. The Concurrent C source code for the stochastic distribution function appears in Appendix F (distrib.cc).

4.4.3. The Logical Processes of the Distributed Simulator

The logical processes of the distributed simulation perform the actual simulation. The logical processes send and receive timestamped messages that represent the jobs in the physical queueing network system. In the distributed simulator, logical processes are implemented as Concurrent C *processes*. To reduce the number of Concurrent C processes in the simulation, the operation of the merge points was incorporated into the queue, sink, and branch processes.

The processes in the distributed simulator use the Waiting Rules described in Chapter 2.2 to maintain causality. The first Waiting

Rule is already provided by Concurrent C in the form of <u>transactions</u>. Concurrent C <u>transactions</u> make a process wait until the message it has sent is accepted (received) before allowing the process to continue. The Concurrent C source code for implementing the second Waiting Rule is shown in Figure 4.4. This section of code makes a process wait until there is a message on every possible incoming line before accepting the message with the smallest timestamp.

```
/* Iam.Send is a transaction pointer to the Send transaction */
/* Num_In is the number of incoming lines                    */
/* Msg.Timestamp is the timestamp of the message (Msg)       */

select {
    accept Send(Msg)
        suchthat(c_transcount(Iam.Send) == Num_In)
        by(Msg.Timestamp)
    { /* Process The Message Msg */ };
or
     /* ... other transactions and alternatives ... */
}
```

Figure 4.4: Concurrent C Code for Receiving Incoming Messages

The <u>suchthat</u> and <u>by</u> clauses of the <u>accept</u> statement and the c_transcount() function provided by Concurrent C makes the implementation the second Waiting Rule very easy. The c_transcount() function returns the number of outstanding messages waiting at the transaction specified by the function. The <u>suchthat</u> clause prevents the transaction from being accepted until the suchthat expression is true. Once the suchthat expression is true, the <u>by</u> clause sorts the outstanding messages and selects the message with the smallest value that is specified in the by expression. The code section is Figure 4.4 illustrates of operation of a merge point such as the one shown in Figure 3.2. This code section is incorporated into the queue, sink, and branch processes to eliminate the need for a merge process.

4.4.3.1. Source

Figure 4.5 shows the general algorithm for the source process. The source uses a stochastic distribution function to generate an arrival time. The arrival time is used to update the source's local simulation clock. The source then generates a message using the value of the updated simulation clock as the timestamp and sends the message. The source process keeps track of the average arrival time and the number of simulated jobs generated, and reports these statistics to the collector. The Concurrent C source code for the source process appears in Appendix F (source.cc).

```
Process Source
Begin
    Time  = 0.0
    Accept setup()
    Loop
        Time = Time + arrival_time()
        Msg.Message   = { Some message }
        Msg.Timestamp = Time
        Send(Msg)
        { Record statistics }
    End Loop
End Process Source
```

Figure 4.5: General Algorithm for the Source Process

4.4.3.2. Queue

Figure 4.6 shows the general algorithm for the queue process. A queue process maintains a fixed size queue buffer as specified by the input model. The queue buffer is implemented as a linked list. Incoming messages are always attached to the tail of the list. Dequeueing (i.e. removing a message for the queue buffer) is determined by a queueing discipline specified by the input model. There are four queueing disciplines implemented in the queue process: FIFO

(First In, First Out), LIFO (Last In, First Out), SIRO (Service In Random Order), and PRIO (Priority).

```
Process Queue
Begin
    Accept setup()
    Create a Que
    Loop
        Select
          (Que NOT Full) :
             Accept Send(Msg)
                suchthat (There are Num_in messages)
                by (smallest (Msg.Timestamp))
             { Put Msg in Que }
        Or
          (Que Not Empty) :
             Accept Request ()
             { Get Msg from Que by Discipline
               TReturn Msg
             }
        End Select
        { Record statistics }
    End Loop
End Process Queue
```

Figure 4.6: General Algorithm for the Queue Process

The messages sent to the queue by other logical process are put into the queue's queue buffer, unless the queue buffer is full. If the queue's queue buffer is full, the queue process will not accept any incoming messages until a message is dequeued. The server associated with the queue process requests messages from the queue. These requests are granted unless the queue buffer is empty, in which case the server must wait until a message becomes available in the queue buffer. Because of this close relationship with the associated server, the queue process is always created on the same processor as its associated server.

The queue process handles NULL messages by keeping a special NULL message queue buffer. This "NULL queue buffer" holds only one NULL

message. Whenever a queue receives a NULL message, it puts the NULL message in the NULL queue buffer, which may overwrite a previously received NULL message. This helps reduce the number of NULL messages in the system. When the server requests a message and there is a NULL message in the NULL queue buffer, the queue will give the server the NULL message if the regular queue buffer is empty. If, however, there is a normal message in the regular queue buffer the queue will dequeue and give the server a normal message and discard the NULL message if the timestamp of the normal message is larger then the timestamp of the NULL message.

The queue keeps track of the average time a message was in the queue buffer and the average and the current number of messages in the queue buffer. The queue process reports these statistics to the collector. The Concurrent C source code of the queue processes appears in Appendix F (queue.cc).

4.4.3.3. Server

Figure 4.7 shows the general algorithm for the server process. The server requests messages from its associated queue and waits until the request is granted and a message is returned. The server then uses a stochastic distribution function to generate a service time. The service time is used to update the server's local simulation clock and the timestamp of the message. The updated message is then sent. The server keeps track of the average service time, the number of simulated jobs serviced, and the utilization of the server (i.e. percent busy).

The server processes services NULL messages in the same fashion

as normal messages are serviced by generating a service time, updating the timestamp, and sending the NULL. The server, however, does not record statistics for NULL messages. The Concurrent C source code of the source process is given in Appendix F (source.cc).

```
Process Server
Begin
    Time = 0
    Accept setup()
    Loop
        Msg = Request(Msg)
        Time = Maximum(Time, Msg.Timestamp) + service_time()
        Msg.Timestamp = Time
        Send(Msg)
        { Record statistics }
    End Loop
End Process Server
```

Figure 4.7: General Algorithm for the Server Process

4.4.3.4. Sink

Figure 4.8 shows the general algorithm for the sink process. The sink process receives messages and discards them. The sink, however, keeps track of the number of discarded messages, excluding NULL messages, and reports this statistic to the collector. The Concurrent C source code for the sink process appears in Appendix F (sink.cc).

```
Process Sink
Begin
    Accept setup()
    Loop
        Accept Send (Msg)
           suchthat (there are Num_In messages)
           by (smallest (Msg.Timestamp))
        { Discard Msg }
        { Record statistics }
    End Loop
End Process Sink
```

Figure 4.8: General Algorithm for the Sink Process

4.4.3.5. Branch

Figure 4.9 shows the general algorithm for the branch process. When a branch process receives an incoming message, it selects an outgoing line and sends the message on that line. NULL messages are processed in the same manner. The actual Concurrent C source code for the branch process is given in Appendix F (branch.cc).

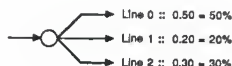
```
Process Branch
Begin
    Time = 0
    Accept setup()

    Loop
        Select
           Accept Get_Time()
           { TReturn Time }
        Or
           Accept Send (Msg)
              suchthat (there are Num_In messages)
              by (smallest (Msg.Timestamp)
           {   Time = Msg.Timestamp
               { determine on which line to send message
                 and send the message. (See Figure 4.11)
               }
           }
        End Select
    End Loop
End Process Branch
```

Figure 4.9: General Algorithm for the Branch Process

The selection of an outgoing line is based on probabilities. Each outgoing line of a branch has a predetermined probability of being selected (Figure 4.9) as specified by the input model. The sum of the probabilities of the outgoing line must total one (1.0). The branch process uses the probabilities to calculate a "range" for each outgoing line (Figure 4.9). The branch selects an outgoing line by generating a uniform random variable (X) and determining the range in which X falls (Figure 4.9). The general algorithm used by the branch

to select an outgoing line based on probabilities is shown in Figure 4.10.



Line	Probability	Range	Comments
0	0.5	0.0 <= X < 0.5	Select Line 0
1	0.2	0.5 <= X < 0.7	Select Line 1
2	0.3	0.7 <= X < 1.0	Select Line 2

Figure 4.10: Probability of Selecting an Outgoing Line

The branch processes do not collect any statistical information, but the branch processes are very important in the resolution of deadlock. The branch processes aid deadlock resolution by providing a "Get_Time" transaction. The Get_Time transaction allows a deadlock resolver process to obtain the current simulation time (i.e. the timestamp of the last message sent) of a branch process.

```
  /* prob[i] is probability of selecting line i */
  /* line[i] is outgoing line i                 */
  /* Num_Out is number of outgoing lines        */

X = { Generate a Uniform Random variable in the
      range 0.0 < X < 1.0 }
low = 0.0
For i = 0 to Num_Out Do
    If (low <= X < prob[i] + low) Then
        Send(Msg) on Line[i]
        Exit For
    End If
    low = low + prob[i]
End For
```

Figure 4.11: General Algorithm for Selecting an Outgoing Line

4.5. Deadlock Detection and Resolution

We have implemented a deadlock detection and resolution strategy to cope with the problems of deadlock. The deadlock detection and

resolution strategy is composed of a deadlock detection mechanism implemented in the kernel of Concurrent C [HAMM88] and a deadlock resolution mechanism implemented at the simulator level. The deadlock detection mechanism is implemented as several "kernel level" processes called "deadlock detectors." The deadlock resolution mechanism is implemented as several "simulator level" processes called "deadlock resolveers." A deadlock detector and a deadlock resolver are started on every processor involved in the simulation.

4.5.1. The Deadlock Detection Mechanism

The deadlock detection mechanism was implemented as part of the kernel of Concurrent C [HAMM88]. The deadlock detection mechanism was implemented at the kernel level because process state information is readily available at the kernel level. The kernel level deadlock detectors use the process state information to determine if a knot of deadlocked processes exists. If a deadlock detector finds a knot of deadlocked processes, such as the one shown in Figure 3.4, it constructs a list of the processes involved in the deadlock (LP11, LP12, LP13, LP14, LP15, LP16) and reports the list to a deadlock resolver process.

The deadlock detector processes are capable of detecting both "local" and "global" deadlock [HAMM88]. Local deadlock consists of a knot of deadlocked processes on a single processor, whereas global deadlock consists of a knot of deadlocked processes distributed across several processors. Figure 4.12 shows an example of both local and global deadlock, and local and global deadlock detection.

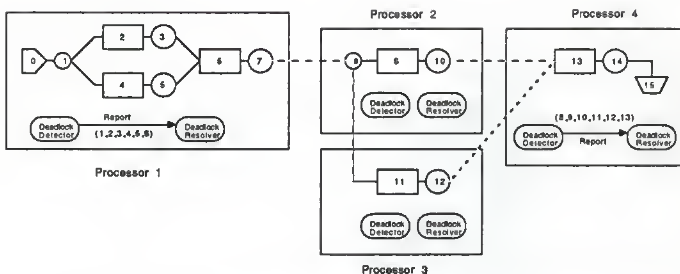


Figure 4.12: An Example of Local and Global Deadlock and Local and Global Deadlock Detection

Local deadlock occurs on Processor 1 and is detected by the deadlock detector on Processor 1. The deadlock detector constructs a list of the processes involved in the deadlock (1, 2, 3, 4, 5, 6) and reports the list to the deadlock resolver on Processor 1. Global deadlock is distributed across Processors 2, 3, and 4. In this example, the deadlock detector on Processor 4 detects the global deadlock. The deadlock detector then constructs a list of the processes involved in the deadlock (8, 9, 10, 11, 12, 13) and reports the list to the deadlock resolver on Processor 4. As shown in Figure 4.12, the deadlock detector processes can simultaneously detect and report multiple occurrences of deadlock.

A further explanation of the deadlock detection algorithm and the deadlock detection mechanism appear in [HAMM88].

4.5.2. Deadlock Resolution Mechanism

The deadlock resolution mechanism is implemented as part of the distributed simulator. We developed an algorithm for the resolver

processes that is capable of resolving both the local and the global deadlock reported by the kernel level deadlock detectors. Our algorithm does not require the complex distributed computations found in the deadlock detection and resolution strategy described in Chapter 3.3.3, but instead, uses information provided by the kernel to resolve deadlock. Our algorithm resolves deadlock by sending NULL messages, similar to the ones used by the deadlock avoidance strategy described in Chapter 3.3.1. Figure 4.13 shows the general algorithm we used to implement the resolver processes.

```
Process Resolver:
    Accept setup()
    Loop
        Accept report(List)
        { Deadlocked_List = List }

        For { each Branch (B) in the Deadlocked_List } Do
            State = query_state( B )
            If State == Sending Then
               Msg = intercept_transaction ( B )
               Time = Msg.Timestamp
            Else /* State == Accepting */
               Time = B.Get_Time()
            End If
            Res_Msg.Message   = NULL
            Res_Msg.TimeStamp = Time
            For { each outgoing line (L) in B } Do
               If B is not sending on L Then
                  Res_Send(Res_Msg) on line L
               End If
            End For
        End For
    End Loop
End Process Resolver
```

Figure 4.13: General Algorithm for the Deadlock Resolver Process

The deadlock resolver waits until a kernel level deadlock detector reports deadlock. A deadlock report takes the form of a list of deadlocked process sent to the resolver. We call this list the "Deadlocked_List." The resolver then searches the Deadlocked_List for

occurrences of branch processes.

When the resolver finds a branch process in the Deadlocked_List it calls the query_state() function to get the process state of the branch. A branch process is either in an "Accepting" state (waiting for a message to arrive) or in a "Sending" state (waiting for a sent message to be accepted). The query_state() function is provided as part of the modified kernel of Concurrent C to aid in deadlock resolution [HAMM88]. The resolver uses the process state of the branch to determine the method for obtaining the timestamp of the last message sent by the branch.

If the branch is in a Sending state, the timestamp of the last message is contained in the message that the branch is currently sending. The resolver, therefore, calls the intercept_transaction() function to obtain a "copy" of the message. The intercept_transaction() function is also provided as part of the modified kernel to aid in deadlock resolution [HAMM88]. Once the resolver has a copy of the message, it can obtain the timestamp. However, if the branch is in an Accepting state, the resolver must use the branch's Get_Time transaction because there is no message to intercept. The value returned by the Get_Time transaction is, however, the same as the timestamp of the last message sent by the branch.

The resolver uses the timestamp of the last message sent as the timestamp for a NULL message. A copy of this NULL message is sent on every unused outgoing line of the branch to resolve deadlock. An unused outgoing line is defined as any line on which the branch is not currently sending a message. To prevent the resolver from being blocked or even deadlocked itself, the resolver sends the NULL

messages as a asynchronous message. Asynchronous message passing is described in Appendix B.

Since Concurrent C does not allow both synchronous and asynchronous messages to arrive at the same transaction, an async transaction called "Res_Send" was used. Figure 4.14 shows the Concurrent C code used to implement the receiving synchronous "Send" and asynchronous "Res_Send" messages. This code section is used by the queue, sink, and branch logical processes to receive incoming messages and NULL messages sent by a deadlock resolver without violating causality.

The code in Figure 4.14 operates similarly to the code shown in Figure 4.4. The Send transaction accepts an outstanding Send message when the number of outstanding Send messages is the same as the number of incoming lines and there are no outstanding Res_Send messages. If, however, there are outstanding Res_Send messages, the Res_Send transaction waits until the number of both the outstanding Send and Res_Send messages is the same as the number of incoming lines before accepting the Res_Send message with the smallest timestamp. The Res_Send transaction then discards all other outstanding Res_Send messages and determines if there is an outstanding Send message with a smaller timestamp. If there is, the Res_Send transaction accepts the Send message with the smallest timestamp and processes it, otherwise the Res_Send message is processed.

```
/* Iam.Send: transaction pointer to transaction Send         */
/* Iam.Res_Send: transaction pointer to transaction Res_Send */
/* Num_In: the number of incoming lines                      */
/* Msg.Timestamp:  the timestamp of the message (Msg)        */

select {
    accept Send(Msg)
        suchthat(c_transcount(Iam.Send) == Num_In AND
                 c_transcount(Iam.Res_Send) == 0)
        by(Msg.Timestamp)
    { /* Process The Message "Msg" */ };
or
   accept Res_Send(Msg)
        suchthat(c_transcount(Iam.Send) +
                 c_transcount(Iam.Res_Send) == Num_In
        by(Msg.Timestamp)
    { Res_Msg = Msg; };
    while (c_transcount(Iam.Res_send) > 0)
       accept Res_Send(Msg) { /* Discard Msg */ };
   select {
      accept send(Msg)
         suchthat (Msg.Timestamp < Res_Msg.Timestamp)
         by (Msg.Timestamp)
         { The_Message = Msg; };
   or
      The_Message = Res_Msg;
   }
   /* Process The Message "The_Message" */

or
     /* ... other transactions and alternatives ... */
}
```

Figure 4.14: Concurrent C Code for Receiving Send and Res_Send Messages

A NULL messages received by a logical process is processed and sent to the next logical process in the simulation. Eventually the NULL message will arrive at the "merge point" and resolve the deadlock. Figure 4.15 shows an example of deadlock resolution using the intercept_transaction() function. In this example, the merge point at LP6 is waiting for a message from LP5. The server (LP5) is not able to send a message because the branch (LP1) is blocked trying to send to LP2 and the queue (LP4) is empty. The deadlock detector detects the deadlock and reports the list of deadlocked processes

(LP1, LP2, LP3, LP4, LP5, LP6) to the resolver. The resolver finds the branch (LP1) in the list and determines that LP1 is in a Sending state. Therefore, the resolver uses the intercept_transaction() function to get a copy of the message (M,82) that LP1 is sending. The resolver then sends the NULL message (NULL,82) on the other outgoing line of the branch. The NULL message is propagated through LP4 and LP5, and the updated NULL message (NULL,85) arrives at LP6, thus resolving the deadlock.

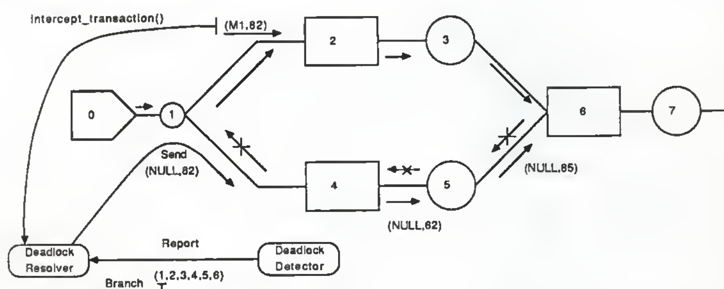


Figure 4.15: Deadlock Resolution using intercept_transaction()

Figure 4.16 shows an example of deadlock resolution using the Get_Time transaction. The queue (LP88) is empty and is waiting for a message to arrive from LP90. The branch (LP90) is waiting for a message from LP89 and the server (LP98) is waiting for LP88 to grant its request. The deadlock detector detects the deadlock and reports the list (LP88, LP89, LP90) to the resolver. The resolver finds the branch (LP90) in the list and determines that LP90 is in an Accepting state. Therefore, the resolver uses the Get_Time transaction to obtain the current simulation time (1988) of the branch. The resolver con-

structs the NULL message (NULL, 1988) and sends a copy of the NULL message on each outgoing line of LP88. One of the NULL messages arrives at LP88 and resolves the deadlock.

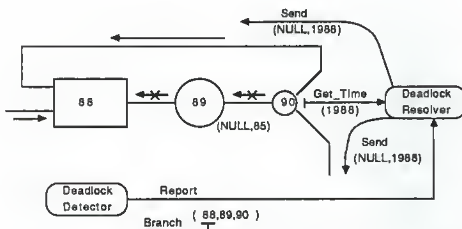


Figure 4.16: Deadlock Resolution using the Get_Time transaction

A deadlock detector may at times report nonexistent deadlock. Nonexistent deadlock is called "phantom deadlock" and can occur when the deadlock detector detects the deadlock that the resolver is currently resolving. Our deadlock resolution algorithm does not concern itself with phantom deadlock, because phantom deadlock rarely occurs and the resolution of phantom deadlock does not harm the simulation. The deadlock resolver processes resolves phantom in the same fashion as it resolves real deadlock. The only effects of resolving phantom deadlock are that unnecessary NULL messages are generated and sent.

The deadlock resolution mechanism is only concerned with resolving simulation deadlock. The deadlock detector mechanism, however, is capable of detecting both simulation and model deadlock. Therefore, the deadlock resolution mechanism should provide a mechanism for distinguishing simulation deadlock from model deadlock. Our deadlock resolver, however, does not currently identify model deadlock.

One method that could be used to identify model deadlock involves keeping a "history list" of past deadlocks. This history list would be composed of entries containing the Deadlocked_List and the timestamp of the NULL message sent to resolve the deadlock. The resolver could detect model deadlock by comparing current (i.e. the most recently reported deadlock) Deadlocked_List and the timestamp of the current NULL message with entries in the history list. If the resolver finds an entry in the history list that matches the current Deadlocked_List and timestamp, the resolver can "suspect" model deadlock. The resolver must, however, take into account that the suspected model deadlock may actually be an occurrence of phantom deadlock. The solution to this problem is to look for multiple entries that match the current information, and if such entries are found, then model deadlock would be detected.

Since model deadlock cannot be resolved, the deadlock resolver could report the occurrence of the model deadlock to the collector. The collector would then inform the graphics front-end that model deadlock has occurred and terminate the simulation.

The implementation of alternative kernel level deadlock detection mechanisms [HAMM88] may require the implementation of an alternative deadlock resolution mechanism, particularly if the deadlock detection mechanisms cannot provide a list of the deadlocked processes. Our current deadlock resolution algorithms requires that the deadlock detection mechanism provide the list of deadlock process, but we have developed an alternative algorithm that does not require a Deadlocked_List from the deadlock detector.

Our alternative algorithm takes a list of logical processes

provided by distributed simulator and constructs a new list of only the branch processes. The deadlock resolver waits until a deadlock is reported and then treats the list of branch processes as if it were the Deadlocked_List. Our algorithm basically resolves a reported deadlock by sending a NULL message on every unused outgoing line of every branch process in the simulation. Our algorithm tends to flood the simulation with NULL message, but deadlock is resolved. The Concurrent C source code for this algorithm is given in Appendix F (resolver.cc).

5. Summary and Future Research

5.1. Summary

We have implemented a distributed discrete-event simulator in the distributed programming language Concurrent C. Our distributed simulator and the graphics front-end make up a distributed simulation environment. The distributed simulation environment allows a wide variety of queueing network models to be created and simulated. The simulation of these queueing network models is conducted by the distributed simulator on the loosely coupled multiprocessor computer system discussed in Appendix A.

Our distributed simulator uses a deadlock detection and resolution strategy to cope with the problems of deadlock that may occur during the course of the distributed simulation. We developed the deadlock resolution mechanism that is capable of resolving both the local and global deadlock states reported by the kernel level deadlock detection mechanism. Our deadlock resolution algorithm uses several kernel functions and NULL messages to resolve the reported deadlocks.

Our distributed simulator was implemented in 2700 lines of Concurrent C source code, of which only 150 lines were devoted to the deadlock resolver. The Concurrent C programming language greatly simplified the task of developing the distributed simulator. The message passing facilities provided by Concurrent C allow the Waiting Rules (Chapter 3.1) used by the logical processes to be easily and cleanly implemented.

We have shown that deadlock detection in the kernel of a distributed programming language and deadlock resolution at the application

level is a valid approach to distributed simulation. Because deadlock detection was implemented in the kernel of Concurrent C, the deadlock detector processes can use the process state information provided by the kernel to rapidly detect deadlock and produce a list of deadlocked processes. Furthermore, the deadlock detection mechanism can provide kernel level deadlock resolution assistance facilities that will aid in deadlock resolution. The deadlock resolution mechanism, implemented at the application level, can readily access information that is only available at the application level. The deadlock resolver uses this information and the information provided by the deadlock detector and its resolution assistance facilities to resolve deadlock. The combination of these two mechanisms allows our distributed simulator to quickly detect and resolve the deadlocks that interrupt a distributed simulation of a queueing network model.

5.2. Problems

One of the major problems encountered in the implementation of the distributed simulator was the discovery of a "bug" in the AT&T C compiler (version 4.1). Concurrent C uses the C compiler during the final phases of program compilation. The bug manifest, whenever a structure of 4 K-bytes (4096 bytes) or larger is passed as a parameter or a message. The bug causes the distributed simulator to either "hang" or "core dump." The bug was reported and certified by AT&T Software Support and, according to them, will be corrected in version 4.3 of the C compiler. In the mean time, the sizes of parameters and messages are limited to less than 4 K-bytes.

5.3. Future Enhancements

Many enhancements that could be made to improve the distribed simulator. One enhancement is the implementation of new simulation elements such as resource allocators and deallocators. A second enhancement involves attaching job variables to the messages and implementing routing control mechanisms within the logical processes. The routing control mechanisms would use the job variables to determine the path the message would follow through the simulation instead of leaving the decision to chance. A third enhancement involves modifying our deadlock resolution algorithm to identify and cope with model deadlock.

6. Future Projects

There are several future projects related to the distributed simulator. One project could involve comparing the operation of our distributed simulator that uses a deadlock detection and resolution strategy with the operation of a distributed simulator implemented using a deadlock avoidance strategy and one implemented using a "Time Warp" strategy. A second project could involve comparing the operation of our simulator on our loosely coupled computer system with the operation of our simulator on a tightly coupled multiprocessor computer system. A third project could examine the application of the distributed simulation of "Petri Networks."

References

[AT&T88] Enhanced TCP/IP WIN/3B, The Wollongong Group, Inc., 1987.

[BACH86] M. J. Bach, The Design of the UNIX Operating System, Prentice-Hall, 1986.

[BRUE80] S. C. Bruell, G. Balbo, Computational Algorithms for Closed Queueing Networks, North-Holland, 1980.

[BRYA79] R. E. Bryant, "Simulation on a Distributed System," Distributed Computing Conference, 1979, pp. 544-552 .

[BUTLER] J. Butler, (unpublished notes), Kansas State University, 1988.

[CHAN78] K. M. Chandy & J. Misra, "A Nontrival Example of Concurrent Processing: Distributed Simulation," IEEE COMPSAC 78, 1978, pp. 822-826.

[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.

[FRAN77] W. R. Franta, The Process View of Simulation, North-Holland, 1977.

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

[GEHA88b] N. H. Gehani, "Message Passing: Synchronous versus Asynchronous", AT&T Bell Laboratories, (Submitted for Publication).

[GHOS84] J. Ghosh, "Asynchronous Simulation of Some Discrete Time Models," Proc. Winter Simulation Conference, November 1984, pp. 467-469.

[GHOS85] J. Ghosh, "Asynchronous Simulation of Discrete Event Simulation," Proc of the 18th Annual Simulation Symposium, 1985, pp. 255-263.

[HALL88] M. Hall, Simulating a Distributed File System With Various Types of Networks, Masters Report, Kansas State University, 1988.

[HAMM88] S. Hammond, Distributed Deadlock Detection in Concurrent C Masters Thesis, Kansas State University, 1988.

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

[MISR86] J. Misra, "Distributed Discrete-Event Simulation," Computing Surveys, Vol. 18, No. 1, March 1986, pp. 39-65.

[PEAC79] J. K. Peacock, J. W. Wong, E. G. Manning, "Synchronization of Distributed Simulation Using Broadcast Algorithms," Proc. of the Winter Simulation Conference, December, 1979, pp. 3-10.

[PETE83] J. Peterson, A. Silberschatz, Operating System Concepts, Addison-Wesley Publishing Co., 1983.

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

[STAM85] R. A. Stammers, "Ada on Distributed Systems," Concurrent Languages in Distributes Systems, G. L. Reijns and E. L. Dagless (eds), Elsevier Science Publishers (North Hollond), 1985.

[UNIX86] UNIX Programmer's Supplementary Documents, Volume 1, USENIX, 1986.

[WAIT87] M. Waite (editor), UNIX Papers for UNIX Developers and Power Users, Howard W. Sam & Company, 1987.

Appendix A: Multiprocessor Computer Systems

Multiprocessor computer systems can be divided into two categories: tightly coupled systems and loosely coupled systems [WAIT87]. A tightly coupled multiprocessor computer system is composed of two or more processors that share the same physical memory. The processors of a tightly coupled multiprocessor computer system are generally on the same machine and are connected by a common bus. Figure A.1 shows an example of a tightly coupled multiprocessor computer system. The VAX 8650 and the CRAY XMP are two examples of tightly coupled multiprocessor computer systems.

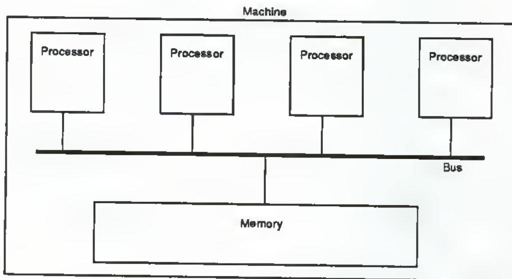


Figure A.1: A Tightly Coupled Multiprocessor Computer System

A loosely coupled multiprocessor computer system is composed of two or more processors that do not share the same physical memory. A loosely coupled multiprocessor computer system is generally made up of several uniprocessor computers connected by a network. Figure A.2 shows an example of a loosely coupled multiprocessor computer system. A network of minicomputers is an example of a loosely coupled multiprocessor computer system.

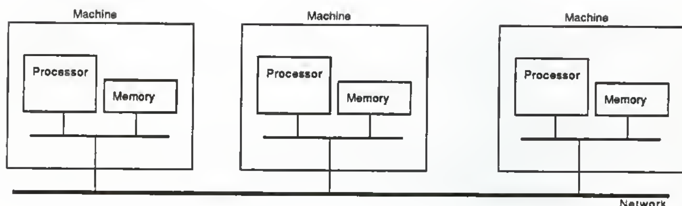


Figure A.2: A Loosely Coupled Multiprocessor Computer System

The distributed simulation environment described in chapter 4 runs on a loosely coupled network of minicomputer operated by the Computing and Information Sciences Department at Kansas State University. This network (KSU.CIS Network) is composed of AT&T 3B2/400, 3B2/310, and 3B15 minicomputers, and Xerox 1186 graphics work stations. The graphics front-end of the distributed simulation environment runs on a Xerox 1186 graphics work station and the distributed simulator runs the AT&T 3B minicomputers. The distributed simulator runs on the AT&T 3B minicomputers because the processes of Concurrent C can only be distributed on a homogeneous multiprocessor computer system. The AT&T 3B minicomputers are homogeneous because they are object code compatible. Figure A.3 shows the distributed simulation environment conducting a distributed simulation on a section of the KSU.CIS Network. The names of the minicomputers (i.e. november, hotel, echo, and phobos) represent the "host" names of the minicomputers.

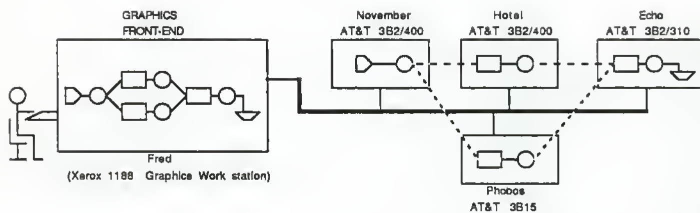


Figure A.3: The Distributed Simulation Environment on the KSU.CIS Network of Minicomputers

Appendix B: Message Passing

Message passing facilities are used in distributed programs for interprocess communication and synchronization. These facilities can be classified into two categories: synchronous (blocking) and asynchronous (non-blocking). In the case of a synchronous message send, the sender waits (blocks) until the receiver has accepted the message. On the other hand, in the case of an asynchronous message send, the sender continues execution after sending the message, without waiting for the receiver to accept it. The difference between synchronous and asynchronous message receives is similar [GEHA88b].

Concurrent C provides both synchronous (called _transactions_) and asynchronous (called _async transactions_) message passing facilities. A description of these facilities appears in [GEHA88] and a comparison of the two message passing facilities appears in [GEHA88b]. The synchronous message passing facilities are similar to Ada's _extended rendezvous_ which allows bidirectional communication between the interacting processes. The asynchronous message passing facilities, on the other hand, only allows unidirectional communication between the interacting processes.

The logical processes of the distributed simulator use synchronous message passing to send their normal messages. The deadlock resolver processes use asynchronous message passing to send NULL messages, thus preventing the resolver processes from becoming blocked or even deadlocked themselves. The main process of the distributed simulator uses asynchronous message passing to distribute the process id's and other "setup" information provided by the input model (as

described in chapter 4) to the logical processes and the terminator process uses asynchronous message passing to send terminate messages.

Appendix C : Graphics Interface

The communication connection between the graphics front-end and the distributed simulator is established by a special "connection program" (Appendix F, connect.c). This program opens a "TCP/IP socket port" [AT&T88] and waits for graphics front-end to connect to the port. Once the graphics front-end has connected, the connection program starts the distributed simulator. The reasons for using the connection program is to resolve a naming conflict between the TCP/IP accept() function and the Concurrent C accept statement. The TCP/IP socket provides a reliable bidirectional network communication link between the graphics front-end and the distributed simulator. TCP/IP sockets are described in [AT&T88], [BACH86], and [UNIX86]. The communication connection between the graphics front-end and the distributed simulator is used to transfer the input model from the graphics front-end to the distributed simulator and the collective statistics report (Chapter 4) from the distributed simulator to the graphics front-end.

C.1: Format of the Input Model

The "input model" is a numeric representation of the queueing network model. The information contained within the input model will tell the distributed simulator how to construct and distributed the queueing network model. The distributed simulator can either obtain the input model from the socket connection or from a file. If a socket connection is used, the connection program establishes the socket connection and starts the simulator. If, however, the input model is to come from a file, the distributed simulator is started directly. The file contains the same information that would have been

sent over the socket connection. Figure C.1 shows the BNF notation for the input model and Figure C.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> <Num_In>
                 <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 C.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 C.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 (described in Chapter 4).

<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
               (described in Appendix D).

<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 C.2: Description of the BNF Non-Terminals in Figure C.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 C.3: List of Machines used in Figure C.2

Figure C.3 shows a list of the computers that make up the loosely coupled multiprocessor computer system on which the distributed simulator runs (Appendix 1). The following list contains the machine numbers used by the input model, the host name of the computer, and the model of the computer. All of the following computer are products

of AT&T.

Figure C.4 shows an example of the input model of the queueing network in Figure A.3.

```
( ( 99 0 ) )
( ( 0 0 ) ( 2 0 0 50.0 ) 8 0 100 1 )
( ( 3 1 ) 8 0 1 2 ( ( 2 0.60 ) ( 3 0.40 ) )
( ( 2 2 ) ( 4 0 0 50.0 ) 2 0 4 10 0 1 )
( ( 2 3 ) ( 4 0 0 50.0 ) 15 0 4 10 0 1 )
( ( 2 4 ) ( 4 0 0 50.0 ) 14 0 5 10 0 2 )
( ( 1 5 ) 14 0 1 )
( ( 99 1 ) )
( ( 98 0 ) 0 200 )
```

Figure C.4: Example of an Input Model

C.2: Format of the Collective Report

The collector process collects individual status reports from each logical process and composes a collective statistics report. This report is sent to the graphics front-end over the socket connection. Figure C.5 shows the BNF of the collective statistics report and Figure C.6 gives a description of the BNF nonterminals. If, however, the distributed simulator is obtaining the input model from a file, the collective statistics report is displayed on the console.

```
<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 C.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_8usy> :: 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 C.6: Description of the 8NF Non-Terminals in Figure C.5

Figure C.7 shows an example a collective statistics report of the queueing network described in Figure C.4. This report occurred at simulation time 1200.

```
(1200.0000)
(0 76)
(2 38.7461 0.0000 0 12)
(3 34.8273 0.0000 0 12)
(4 88.4272 40.0000 4 13)
(5 13)
($$)
```

Figure C.7: Example of a Collective Statistics Report

Upon receiving the collective statistics report, the graphics front-end will graphically display the report. The user can then elect to continue or terminate the simulation. In either case, the graphics front-end will send a simulation control message back to the distributed simulation. Figure C.8 shows the format of the simulation control messages. Upon receiving a simulation control message the

distributed simulator will either continue or terminate the simulation.

```
"( ( 97 0 ) )"  ::  simulation continue control message
"( ( 96 0 ) )"  ::  simulation termination control message
```

Figure C.8: Simulation Control Messages

Appendix D: Stochastic Distributions Functions

The distributed simulator uses several stochastic distribution functions to calculate service and arrival times. The stochastics distribution functions use the drand01() function to obtain a pseudo-random number. The drand01() function returns a floating point value in the range of 0 to 1 (0 <= drand01() < 1). The drand01() function uses the random number generator provided in the C libraries. For the BSD systems, the random number generator is the random() function, which returns a value in the range of 0 to 2,147,483,648 (2^31) (0 <= random() < 2^31). For System V systems, the random number generator is the lrand48() function, which returns a value in the range of 0 to 2,147,483,648 (2^31) (0 <= lrand48() < 2^31). The drand01() functions takes the value returned by the random number generator function and converted it to a floating point value in the range 0 to 1.

Figure D.1 shows the stochastic distribution function used by the distributed simulation. "Type" is a numeric valued used in the input model to identify each stochastic distribution function. "Arg1" and "Arg2" are values used as parameters for the functions.

Type	Name	Arg1	Arg2
0	Fixed	time	
1	Uniform	a	b
2	Poisson	mean	
3	Binomial	n	p
4	Expntl	mean	
5	Normal	mean	stdev
6	Gamma	mean	k
7	Beta	k1	k2
8	Erlang	mean	k
9	Lognormal	mean	stdev
10	Weibull	shape	scale

Figure D.1: Stochastic Distribution Functions

The stochastic distribution functions and their parameters are described in [HALL88]. The source code for the stochastic distribution functions appears in Appendix F, distrib.cc.

Appendix E: Implementation Notes

The distributed simulator has several command line switches that allow debugging and tracing facilities to be activated. These debugging and tracing facilities cause the distributed simulator to display pertinent information about the operation of the distributed simulator. Figure E.1 lists the command line switches and gives a brief description of the function of each switch. Switches #0-16 are used exclusively by the distributed simulator, switches #17-19 are unused, and switches #20-31 are reserved for the deadlock detection mechanism. The debugging and tracing facilities greatly aided the development and debugging of the distributed simulator.

Switch #	Description of Switch
0	tracing for the main process
1	tracing for reading the input model
2	tracing for creating the logical processes
3	tracing for Source logical processes
4	tracing for Sink logical processes
5	tracing for Server logical processes
6	tracing for Queue logical processes
7	tracing for Branch logical processes
8	tracing for the Collector process
9	tracing for Resolver processes
10	tracing for the stochastic distribution functions
11	tracing for the Terminator process
12	tracing of socket communication
13	tracing of the initial socket connection
14	create a statistics report (Stats_Report)
15	create a sizeof report (Size_of)
16	create an "Out Graph" (Out_Graph)

Figure E.1: Command Line Switches

Switch #14 causes the distributed simulator to produce a statistics report (Stats_Report). The Stats_Report is a formatted report of the statistics collected by the collector. Figure E.2 shows a section of a Stats_Report that was produced by the input model in Figure C.4.

```
=============================================================
==  Interval Number: 5     Interval Time: 1200.0000       ==
=============================================================
Status: Normal

Source:  0  Interval: 1200.0000  Simulation Time: 1156.0000
        Inter Arrival Time            Number
     Ave         STD         MAX        Left
  48.167       5.241     61.000          76

Queue/Server:  2  Interval: 1200.0000  Simulation Time: 1158.3556
Queue:  Full:   0.00%  Num Through Queue: 13
      Average Time in Queue            Average in Queue     Num in
     Ave        STD         MAX       Ave        STD        MAX Queue
 579.793   361.080  1009.500     1.154      0.361      2.000     0

Server:   Busy:  38.75%
      Average Time In System         Average Service Time     Number
     Ave        STD         MAX       Ave        STD        MAX  Serviced
  50.135    34.274   128.287    37.401     32.616   128.287        12

Queue/Server:  3  Interval: 1200.0000  Simulation Time: 1104.9237
Queue:  Full:   0.00%  Num Through Queue: 13
      Average Time in Queue            Average in Queue     Num in
     Ave        STD         MAX       Ave        STD        MAX Queue
 359.251   199.393   697.853     1.000      0.000      1.000     0

Server:   Busy:  34.83%
      Average Time In System         Average Service Time     Number
     Ave        STD         MAX       Ave        STD        MAX  Serviced
  37.520    33.533   117.426    32.068     30.544   117.426        12

Queue/Server:  4  Interval: 1200.0000  Simulation Time: 1122.7053
Queue:  Full:  40.00%  Num Through Queue: 14
      Average Time in Queue            Average in Queue     Num in
     Ave        STD         MAX       Ave        STD        MAX Queue
 149.415   114.783   340.605     1.929      1.033      4.000     4

Server:   Busy:  88.43%
      Average Time In System         Average Service Time     Number
     Ave        STD         MAX       Ave        STD        MAX  Serviced
 254.258   172.769   552.041    76.367     68.925   216.002        13

Sink:  5  Interval: 1200.0000  Simulation Time: 1122.7053
  Number Sunk:  13
```

Figure E.2: Stats_Report

Switch #15 causes the distributed simulator to produces a "sizeof" report (Size_of). This report shows the sizes of the various structures used by the distributed simulator. The Size_of report was

used to find structures that were larger then 4 K-bytes. Structures may be larger than 4 K-bytes, if they are not passed as parameters. (The 4 K-byte structure restriction is described in Chapter 5). Figure E.3 shows a sample Size_of report.

```
Size of Standard Types:
int                     4
long                    4
double                  8
char                    1

Values:
MAXPROC                 110
MAXFAN                  5
D_NPROCS                50
D_NPORS                 16

Size of Stochastic Distributions Structures:
struct fixed_rec     8
struct uniform_rec   8
struct poisson_rec   8
struct binomial_rec  12
struct expntl_rec    8
struct normal_rec    16
struct gamma_rec     16
struct beta_rec      16
struct erlang_rec    12
struct lognormal_rec 16
struct weibull_rec   16
struct distrib_rec   40

Size of Parameters:
struct src_param     84
struct sink_param    32
struct srv_param     84
struct q_param       20
struct brn_param     116
struct col_param     2256
struct term_param    2256
struct res_param_rec 5352

Size of Process Table & Data Structures:
struct PS_REC         2252
union  PS_Data_Rec    4
PS_Data_Rec * MAXPROC 440
struct Src_rec        48
struct Q_Srv_rec      56
struct Sink_rec       4
struct fan_rec        12
struct Branch_rec     68
struct queue_list_rec 144

Size of Statistics Structures:
```

```
struct src_stats_rec   44
struct q_stats_rec     64
struct q_srv_stats_rec 140
struct sink_stats_rec  20
struct col_rec         464
struct stats_rec       28
SET                    16

Size of Message:
struct Item_rec        52

Other structures:
struct out_rec         28
struct out_list        3080
struct list_rec        24
CALLERS                24
OUTBUF                 24
```

Figure E.3: Size_of Report

Switch #16 causes the distributed simulator to produce an "Out_Graph" report. The Out_Graph report contains the "ID" of each logical process (the same as the ID from the input model), the process id of the created logical process, and a list of the "ID's" of the logical processes to which the process can send messages. Figure E.4 shows the Out_Graph report produced by the input model from Figure C.4. The "To" field of the Source and Server gives the "ID" of the process to which the Source or Server may send messages. The "#in" field in the Branch, Queue, and Sink gives the number of incoming lines. The "#out" field of the Branch gives the number of outgoing lines and the "To" field of the Branch gives the "ID's" the process to which the Branch may send messages. Process id's for the Collector and the Terminator are also given in the Out_Graph report. The Out_Graph is used to check the correctness of the input model.

```
Source [  0] pid[600002]   To[  1]
Branch [  1] pid[800003]  #in[  1] #out[  2] To [  2] [  3]
Server [  2] pid[A00004]   To[  4]
Queue  [  2] pid[C00005]  #in[  1]
Server [  3] pid[E00006]   To[  4]
Queue  [  3] pid[1000007] #in[  1]
Server [  4] pid[1200008]  To[  5]
Queue  [  4] pid[1400009] #in[  2]
Sink   [  5] pid[160000A] #in[  1]
Collector    pid[180000B]
Terminator   pid[1A0000C]
```

Figure E.4: Out_Graph Report

Concurrent C requires that a copy of the executable distributed simulator be present on each processor involved in the simulation. The path to the executable simulator is compiled into the simulator. Before the simulator is started the executable simulator must be distributed to each processor involved in the simulation. A good future enhancement could be to have the simulator automatically distributed itself. An alternative approach would be to use remote file sharing. Remote file sharing would allow each processor to have access to the same copy of the executable simulator.

Appendix F: Concurrent C Source Code for the Distributed Simulator

F.1: Makefile

```
# Makefile for the Distributed Simulator -- By Edward Vopata
#                                            November 1988
CCC=/usrb/scott/ccc/bin/CCC
CCCLIB=/usrb/scott/ccc/lib/libmpcc50g.a

HDRS=  dclr.h mach.h queue.h $(XTRAH)

XTRAH= set.h item.h stats.h distrib.h proc_param.h proc_table.h \
       proc_spec.h rand.h

X2HDR= Pid.h defs.h spec.h graph.h

SRCS=main.cc build.cc create.cc source.cc queue.cc server.cc sink.cc\
     branch.cc stats.cc set.cc col.cc term.cc distrib.cc sock.cc    \
     sizeof.cc out_graph.cc stats_rpt.cc resolver.cc

OBJS=main..o build..o create..o source..o queue..o server..o sink..o\
     branch..o stats..o set..o col..o term..o distrib..o sock..o    \
     sizeof..o out_graph..o stats_rpt..o resolver..o

# Dealock Detector library
XOBJS= dd.a
# Distributed Simulator
XEQ=Dsim
# Path to the Distributed Simulator
LOC=\"Sim/${XEQ}\"

#For SYS5 systems
CFLAGS= -g -DSYS5 -DSYS5_DIS +M -DDEADLOCK
CCCFLAGS = -g -DSYS5 -DSYS5_DIS -DDEADLOCK
LIB= -lm -lnet

#For BSD systems
#CFLAGS= -g +M
#LIB= -lm

        ${CCC} -c ${CFLAGS} $*.cc

# Make Dsim
${XEQ}: ${HDRS} ${OBJS}
        cc ${CCCFLAGS} -o ${XEQ} ${OBJS} ${XOBJS} ${CCCLIB} ${LIB}
        chmod 755 ${XEQ}

create..o:      ${HDRS} create.cc
        ${CCC} -c ${CFLAGS} -DLOCATE=${LOC} create.cc

# dclr.h is dependent upon XTRAH include files.
dclr.h: $(XTRAH)
        touch dclr.h
```

F.2: dclr.h

```
/*******************************************************************
 * dclr.h -- by Edward Vopata
 *******************************************************************
 *  This include file should be included in all simulation files.
 *  This file contains Defines of LP types, Queuing Methods, Message
 *   types, Stats status, extra process ID's, debugging names,
 *   graphics front-end interface values.  This file also includes
 *   all the other include files in the proper order.  <math.h> is
 *   also included here, therefore this program must be linked with
 *   the math library (-lm) <<< READ THIS >>>.  This file contains
 *   values to provide for Runtime debugging.  To use this insert
 *   printf statements < if (debug[DEBUG_?????]) printf(XXXXX); >
 *   these debugging switches can be set at runtime (on the command
 *   line) and are handled by main().  Warning: debugging messages
 *   are numerous but very helpful in tracking, tracing, and finding
 *   any bugs still crawling around the program
 *  This file also declares some functions used by various functions
 *   in the program.  THIS IS THE MAIN INCLUDE FILE.
 *  The other include files handle specialized definitions.
 *******************************************************************
 */

#define FALSE    0
#define TRUE     1

#define OUT_GRAPH_NAME    "Out_Graph"
#define SIZE_OF_NAME      "Size_Of"
#define STATS_REPORT_NAME "Stats_Report"

    /* Types of logical processes */
#define SOURCE    0    /* Source entity                  */
#define SINK      1    /* Sink entity                    */
#define QUE_SRV   2    /* Queue/Server entity            */
#define BRANCH    3    /* Branch entity                  */
#define MAXTYPE   4    /* Number of different entities */

    /* Maximum number of Logical Process
     * Max number of fan out lines       */
#define MAXPROC  110   /* Max number of processes/entity    */
#define MAXFAN     5   /* Max number of Fan In/Out paths    */

    /* Queuing Disciplines */
#define FIFO      0    /* First In, First Out               */
#define LIFO      1    /* Last In, First Out                */
#define SIRO      2    /* Service In Random Order           */
#define PRIO      3    /* Priority Queueing                 */
#define MAXPRIOR  20   /* Number of Priority values (0-MAX) */

    /* Message Types */
#define MSG_ERR     -1    /* Error msg */
#define MSG_NULL     0    /* NULL msg  */
#define MSG_ITEM     1    /* Item msg  */
```

```
    /* Statistics status */
#define STATS_NORMAL 0    /* Normal Statistics      */
#define STATS_FINAL  1    /* Final Statistics       */
#define STATS_TERM   2    /* Termination Statistics */

    /* Id's for extra processes */
#define RES_ID       (MAXPROC)    /* Resolver Address   */
#define COL_ID       (MAXPROC+1)  /* Collector Address  */
#define TERM_ID      (MAXPROC+2)  /* Terminator Address */

    /* Macro to determine MAX of 2 values */
#define MAX(a,b)  ((a > b) ? a : b )

      /**************** Debugging Switches ****************/
/*      Name           Bit pattern   bit#   Comment                    */
#define DEBUG_MAIN     0x00000001  /* 00:Main: msgs                    */
#define DEBUG_BUILD    0x00000002  /* 01:Build: building msgs          */
#define DEBUG_CREATE   0x00000004  /* 02:Create: process creation      */
#define DEBUG_SOURCE   0x00000008  /* 03:SRC: Source debugging         */
#define DEBUG_SINK     0x00000010  /* 04:SNK: Sink debugging           */
#define DEBUG_SERVER   0x00000020  /* 05:SRV: Server debugging         */
#define DEBUG_QUEUE    0x00000040  /* 06:QUE: Queue debugging          */
#define DEBUG_BRANCH   0x00000080  /* 07:BRN: Branch debugging         */
#define DEBUG_COLLECT  0x00000100  /* 08:Col: collection stats         */
#define DEBUG_RESOLVE  0x00000200  /* 09:Res: Deadlock Resolver        */
#define DEBUG_DISTRIB  0x00000400  /* 10:Stats distributions           */
#define DEBUG_TERM     0x00000800  /* 11:Term: terminator              */
#define DEBUG_SOCK     0x00001000  /* 12:Sock: getline, putline        */
#define DEBUG_DOCNT    0x00002000  /* 13:Connection msgs (Start)       */
#define STATS_REPORT   0x00004000  /* 14:Write stats to file           */
#define SIZE_OF        0x00008000  /* 15:Print Size of Structures      */
#define OUT_GRAPH      0x00010000  /* 16:Print Out Graph of Model      */

/* Switches #17-19: unused */
/* Switches #20-31: reserved for the Deadlock Detection Mechanism */

#define DEBUG_SCT20    0x00100000  /* 20:Scott's Debugging Flags   */
#define DEBUG_SCT21    0x00200000  /* 21:Scott's Debugging Flags   */
#define DEBUG_SCT22    0x00400000  /* 22:Scott's Debugging Flags   */
#define DEBUG_SCT23    0x00800000  /* 23:Scott's Debugging Flags   */
#define DEBUG_SCT24    0x01000000  /* 24:Scott's Debugging Flags   */
#define DEBUG_SCT25    0x02000000  /* 25:Scott's Debugging Flags   */
#define DEBUG_SCT26    0x04000000  /* 26:Scott's Debugging Flags   */
#define DEBUG_SCT27    0x08000000  /* 26:Scott's Debugging Flags   */
#define DEBUG_SCT28    0x10000000  /* 28:Scott's Debugging Flags   */
#define DEBUG_SCT29    0x20000000  /* 29:Scott's Debugging Flags   */
#define DEBUG_SCT30    0x40000000  /* 30:Scott's Debugging Flags   */
#define DEBUG_SCT31    0x80000000  /* 31:Scott's Debugging Flags   */
#define MAX_DEBUG      32

    /* Misc. Defines */
#define MAXLINE       128   /* Max Line Length of Input      */

    /* Types of graphic communication messages */
#define SIG_TERM 96    /* Signal to terminate simulation        */
#define SIG_CONT 97    /* Signal to continue simulation         */
```

```
#define SIG_SOS  98    /* Signal Start of simulation              */
#define SIG_BEGS 99    /* Signal Begin of simulation model data */
#define SIG_ENDS 99    /* Signal End of simulation model data   */

/*******************************************************************/
  /* Include files:
   *  BEWARE: of how these includes are ordered
   *  Reordering them may cause problems.
   *  I originally ordered them by intuition
   */

#include <stdio.h>        /* Standard C include file for FILE & NULL */
#include <math.h>         /* Standard C include for math functions   */
#include "set.h"          /* Set operations                          */
#include "rand.h"         /* Random number Generator Includes.       */
#include "item.h"         /* Inter-Logical Process Includes          */
#include "stats.h"        /* Statistical Includes                    */
#include "distrib.h"      /* Distrib. function for serv & iit times  */
#include "proc_table.h"   /* Process Tables Includes                 */
#include "proc_param.h"   /* Process Input Parameter Includes        */
#include "proc_spec.h"    /* Process Specs Includes                  */
/*******************************************************************/

    /* The largest possible timestamp
     * Max Time Stamp. (HUGE: from <math.h>)
     * MAX_TIME must appear after <math.h> has been included
     */
#define MAX_TIME (HUGE)

    /* Handle different operating systems       */
    /* Don't forget: the "#" must be in Column 1 */
#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();       /* Allocate memory <standard function> */
void   free();          /* Free memory <standard function>     */
void   getline();       /* get a line from a socket            */
void   putline();       /* put a line to a socket              */
```

F.3: proc table.h

```
/*****************************************************************
 * proc_table.h  -- by Edward Vopata
 *****************************************************************
 * Process table definition.
 *****************************************************************
 */

    /* Source data */
struct Src_rec {
   long    num_Gen;          /* Num of items a Source will generate */
                             /* if num_Gen == 0 : Infinite Source   */
   int     Out_ID;           /* Id of LP at end of Outgoing line    */
   struct  distrib_rec dis;  /* Distrib of arrival time of items    */
};

    /* Queue/Server data */
struct Q_Srv_rec {
   int     Q_size;           /* Size of the Queue                  */
   int     Q_method;         /* Method for de-queuing an item      */
   int     Num_fan_in;       /* Number of Incoming lines           */
   int     Out_ID;           /* Id of LP at end of Outgoing line   */
   struct  distrib_rec dis;  /* Distrib of arrival time of items   */
};

    /* Sink data */
struct Sink_rec {
   int     Num_fan_in;  /* Number incoming lines  */
};

    /* Record for dealing with Fanout */
struct fan_rec {
    int ID;          /* Id of LP at end of Outgoing Lines */
    double prob;     /* Probability of taking the line    */
};

struct Branch_rec {
   int     Num_fan_in;                /* Id of Incoming Line       */
   int     Num_fan_out;               /* Number of outgoing lines */
      /* Note: the sum of the Fan_out[i].prob must == 1.00        */
   struct  fan_rec  Fan_out[MAXFAN]; /* Id & prob of out lines   */
};

struct PS_REC {
    int     num_nodes;        /* Number of logical processes  */
    int     Max_Hop_Count;    /* Max. Hop Count               */
    int     type[MAXPROC];    /* Type of logical process      */
    int     mach[MAXPROC];    /* Machine Number               */
    int     virt[MAXPROC];    /* Virtual Processor Number     */

    double sim_term_time;     /* Simulation Termination Time       */
    double stats_interval;    /* Statistics Transmission Interval */

    long   Total_Gen;      /* Total number of Job to be Generated */
```

```
    int    Infinite_Src;   /* TRUE is any source is infinite      */

    int    in_sock;          /* Socket for incoming data */
    int    out_sock;         /* Socket for outgoing data */

    long   debug;           /* Runtime Debugging Flags  */

    process Collector    Col_pid;     /* Pid of Collector  */
    process Terminator   Term_pid;    /* Pid of Terminator */

        /* Variant Record of Process Id's (PID) */
    union {
      process Source Src_pid;     /* Pid of Source */
      process Server Srv_pid;     /* Pid of Server */
      process Sink   Sink_pid;    /* Pid of Sink   */
      process Branch Branch_pid;  /* Pid of Branch */
      long    pid;                /* Generic Pid   */
    } ps_pid[MAXPROC];
    process Queue  Que_pid[MAXPROC];     /* Pid of Queue  */
};

union PS_Data_Rec {
  struct Src_rec     *Src;          /* Source data        */
  struct Q_Srv_rec   *Q_Srv;        /* Queue/Server data  */
  struct Sink_rec    *Sink;         /* Sink data          */
  struct Branch_rec  *Branch;       /* Branch data        */
};
```

F.4: proc_param.h

```
/*****************************************************************
 * proc_param.h  -- by Edward Vopata
 *****************************************************************
 *  process parameters
 *       used by all processes
 *****************************************************************
 */

    /* Short form for transaction pointers */
typedef trans void (*SEND)(ITEM);       /* Normal Messages   */
typedef async trans (*RES_SEND)(ITEM);  /* Resolver Messages */

    /* Parameter list for a Source Logical Process */
struct src_param {
    int    ID;                  /* ID of the Source                  */
    double sim_term_time;       /* Simulation Termination Time       */
    double stats_interval;      /* Statistics transmission Interval  */
    int    Max_Hop_Count;       /* Hop Count for Null Messages       */
    long   num_Gen;             /* Number of Items to Generate       */
    int    Out_ID;              /* ID of LP at end of outgoing line  */
    SEND   send;                /* Transaction pointer to Out_ID     */
        /* Transaction Pointer to Stats. Collector */
    trans  void (*stats) (struct src_stats Src_stats);
       /* Random Distribution record for arrival time */
    struct distrib_rec dis;
    long   debug;       /* Runtime debugging Flags */
};

    /* Parameter list for a Sink Logical Process */
struct sink_param{
    int    ID;                  /* ID of the Sink                    */
    double sim_term_time;       /* Simulation Termination Time       */
    double stats_interval;      /* Stats. transmission Interval      */
    int    Num_Fan_In;          /* Number of incoming lines          */
        /* Trans. pointer to Stats. Collector */
    trans  void (*stats) (struct snk_stats Snk_stats);
    long   debug;               /* Runtime debugging Flags           */
};

    /* Parameter list for a Logical Server Process */
struct srv_param {
    int      ID;                /* ID of the Server                  */
    double   sim_term_time;     /* Simulation Termination Time       */
    double   stats_interval;    /* Stats. transmission Interval      */
    int      Max_Hop_Count;     /* Max Hop Count for Null Messages   */
    int      Out_ID;            /* ID of LP at end of outgoing line  */
    SEND     send;              /* Trans. Pointer to Out_ID          */
        /* Trans. pointer to Stats. Collector */
    trans    void (*stats) (struct srv_stats Srv_stats);
    process Queue Que;          /* PID of corresponding queue        */
    struct   distrib_rec dis;   /* Random Distrib. rec for serv time */
    long     debug;             /* Runtime debugging Flags           */
};
```

```
    /* Parameter list for a Logical Queue Process */
struct q_param {
    int   ID;                   /* ID of the Queue                    */
    int   Q_size;               /* Size of the Queue                  */
    int   Q_method;             /* Method for de-queuing items        */
    int   Num_Fan_In;           /* Number of incoming lines           */
    long  debug;                /* Runtime debugging Flags            */
};

    /* Parameter List of Logical Branch Process */
struct brn_param {
    int     ID;                /* ID of the Branch                       */
    double  sim_term_time;     /* Simulation Termination Time            */
    double  stats_interval;    /* Stats. transmission Interval           */
    int     Max_Hop_Count;     /* Max Hop Count for Null Messages        */
    int     Num_Fan_In;        /* Number of incoming lines               */
    int     Num_Fan_Out;       /* Number of outgoing lines (1-MAXFAN)    */
    struct {
        int    ID;        /* ID of LP at end of outgoing lines      */
               /* Note: the sum of the Fan_Out[i].prob must = 1.00 */
        double prob;      /* Probability of taking that line        */
        SEND   send;      /* Trans. Pointer to ID                   */
    } Fan_Out[MAXFAN];
    long   debug;                  /* Runtime Debugging Flags          */
};

    /* Parameter List of Stats. Collector Process */
struct col_param {
    int    ID;               /* ID of the Collector         */
    struct PS_REC PS;        /* A copy of the Process Table */
};

    /* Parameter List of Terminator Process */
struct term_param {
    int    ID;               /* ID of the Terminator        */
    struct PS_REC PS;        /* A copy of the Process Table */
};
```

F.5: proc_spec.h

```
/*****************************************************************
 * proc_spec.h  -- by Edward Vopata
 *****************************************************************
 * Concurrent C process specifications.
 *****************************************************************
 */
    /*  Process spec definitions */

    /* The Logical Sink Process */
process spec Sink ()
{
   async trans setup (struct sink_param Snk); /* Get Param. List   */
   trans void  send (ITEM item);             /* Incoming messages */
   async trans Res_send (ITEM item);
   async trans term ();                      /* Terminator trans. */
};

    /* The Logical Source Process */
process spec Source ()
{
   async trans setup (struct src_param Src);  /* Get Param. List   */
   async trans term ();                       /* Terminator trans. */
};

    /* The Logical Server Process */
process spec Server ()
{
   async trans setup (struct srv_param Srv);  /* Get Param. List   */
   async trans term ();                       /* Terminator trans. */
};

    /* The Logical Queue Process */
process spec Queue ()
{
   async trans setup (struct q_param Que);   /* Get Param. List    */
   trans void  send (ITEM item);             /* Incoming messages  */
   async trans Res_send (ITEM item);
   async trans term ();                      /* Terminator trans.  */

       /* Get an item from the queu - used by corresponding Server */
   trans void  get_item(ITEM *item, double sim_time);

       /* Get stats from queue - used by corresponding Server      */
   trans struct q_stats_rec stats (double sim_time);
};

    /* The Logical Branch Process */
process spec Branch ()
{
   async trans  setup (struct brn_param Brn); /* Get Param. List   */
   trans void   send (ITEM item);             /* Incoming messages */
   async trans  Res_send (ITEM item);
   async trans  term ();                      /* Terminator trans. */
```

```
   trans double get_time();
};

    /* The Terminator Process */
process spec Terminator ()
{
   async trans setup (struct term_param term); /* Get Param. List  */
   async trans setup1(int num, process anytype Res[20]);
   async trans term ();                          /* Start Termination */
};

    /* The Stats. Collector Process */
process spec Collector ()
{
       /* Collect Incoming Stats from Queue/Server */
   trans void  Que_Srv_stats (struct q_srv_stats_rec q_srv_stats);

       /* Collect Incoming Stats from Sink */
   trans void  Sink_stats (struct sink_stats_rec sink_stats);

       /* Collect Incoming Stats from Source */
   trans void  Src_stats (struct src_stats_rec src_stats);

   async trans setup (struct col_param col);   /* Get Param. List  */
   async trans term ();                        /* Terminator trans */
};
```

F.6: item.h

```
/*********************************************************************
 *  item.h  --  Edward Vopata
 *********************************************************************
 * Definition of the message (item) used in inter-Logical process
 *  communication.  This message contains the time_stamp and the type
 *  of the message.  If the message is Real message then the Priority
 *  field contains the priority of the message, If the message is a
 *  Null msg then the Hop_Count field contains the number of hops
 *  remaining until the Null msg can be discarded (only servers
 *  decrement the Hop Count).  Each Item contains a field that
 *  identifies the the source of the message (sources for real
 *  messages), almost anywhere for Null msgs.  There is also a data
 *  field for additional information (This field is unused at the
 *  current time).  Additional fields may be added to hold routing
 *  information and other useful data.
 *********************************************************************
 */

#define DATA_LEN  10          /* Size of Additional data      */

struct Item_rec {
    double Time_Stamp;        /* Time spamp of the item       */
    double Arrive_Time;       /* Time item arrived in system  */
    double Enter_in_Que;      /* Time Entered current Queue   */
    int    Priority;          /* Priority of item             */
    int    Item_type;         /* Item type (Item or Null Msg) */
    int    Hop_Count;         /* Hop count for Null Msg       */
    int    Id_of_Src;         /* Id of the source             */
    char   data[DATA_LEN];    /* Additional data              */
};

typedef struct Item_rec  ITEM;
```

F.7: queue.h

```
/***********************************************************************
 *  queue.h -- by Edward Vopata
 ***********************************************************************
 * The Queue is handled has a singlely linked list.  Head & Tail
 * pointers are used to maintain the Queue.  A list of unused
 * items is maintained by Free.  When Head == Tail == NULL then
 * the queue is empty.  When Free == NULL then the queue is full.
 * Null messages are maintained in a single item queue.  New incoming
 * Null messages are placed in the Null_Queue & may overwrite an old
 * Null message (this is alright because the new null_msg is
 * guaranteed to have a larger time stamp then the old null_msg.
 * A Null_Msg_in_Q Flag indicates whether there is a Null_msg in the
 * Null_Queue.
 ***********************************************************************
 */

    /* An item in a queue */
typedef struct q_item_rec Q_ITEM;

    /* An item in a queue */
struct q_item_rec {
    ITEM   Item;     /* The item                */
    Q_ITEM *Next;    /* Pointer to the next item */
};


    /* Record for maintaining the queue */
struct queue_list_rec {
    int    ID;              /* Id of Queue/Server                    */
    int    Max_Size;        /* Maximum Size of Queue                 */
    int    Method;          /* Method for de-queuing an item         */
    int    Num_Elem;        /* Number of items in the queue          */
    Q_ITEM *Head;           /* Pointer to the Head of the Queue      */
    Q_ITEM *Tail;           /* Pointer to the Tail of the Queue      */
    Q_ITEM *Free;           /* Pointer to the List of unused Items   */
    Q_ITEM *Free_ptr;       /* Pointer to Start of malloc'ed area    */
    STATS  Que_Time;        /* Time in Queue stats record            */
    STATS  Que_Size;        /* Number in Queue stats record          */
    ITEM   Null_Queue;      /* Null Message Queue                    */
    int    Null_Msg_in_Q;   /* Null_Queue full Flag                  */
};

    /* Short form for the queueing record */
typedef struct queue_list_rec Queue_List;
```

F.8: distrib.h

```
/*********************************************************************
 * distrib.h -- by Edward Vopata
 *********************************************************************
 *  Definition of Stochastic Distribution Functions.  These Functions
 *    are used to generate arrival times and service times.
 *  The functions and their arguments are as follows:
 *         type    name        arg1             [arg2] <-- Optional
 *           0     Fixed       <time:double>
 *           1     Uniform     <lower:long>     <upper:long>
 *           2     Poisson     <mean:double>
 *           3     Binomial    <num:long>       <prob:double>
 *           4     Expntl      <mean:double>
 *           5     Normal      <mean:double>    <stdev:double>
 *           6     Gamma       <mean:double>    <k:double>
 *           7     Beta        <k1:double>      <k2:double>
 *           8     Erlang      <mean:double>    <k:long>
 *           9     Lognormal   <mean:double>    <stdev:double>
 *          10     Weibull     <shape:double>   <scale:double>
 *
 *  Two values min_time and max_time provide truncation for these
 *    functions.  If the values for min_time or max_time == 0.0
 *    then the truncation is ignored for that values.
 *   If (min_time > 0.0 and the distrib value is < min_time) then the
 *     distrib value <- min_time.
 *   If (max_time > 0.0 and the distrib value is > max_time ) then
 *     the distrib value <- max_time.
 *   This will truncated values.  The function get_time generates a
 *     value using a predetermined distribution method.  if the value
 *     is < 0 then the value is re-generated.  Therefore all values
 *     are positive.
 *     (Value >= 0.00).  Time is assumed to be a real (double) value.
 *********************************************************************
 */

   /* Stochastic Distribution Function Types */
#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

   /* Stochastic Distribution Function Parameters */
struct fixed_rec     { double time; };
struct uniform_rec   { long lower, upper; };
struct poisson_rec   { double mean; };
struct binomial_rec  { long trials; double prob; };
struct expntl_rec    { double mean; };
```

```
struct normal_rec    { double mean, stdev; };
struct gamma_rec     { double mean, k; };
struct beta_rec      { double kl, k2; };
struct erlang_rec    { long k; double mean; };
struct lognormal_rec { double mean, stdev; };
struct weibull_rec   { double shape, scale; };

   /* Stochastic Distribution Function Record */
struct distrib_rec {
    int    type;          /* type of distributions           */
    int    debug;         /* Debugging flag <DEBUG_DISTRIB> */
    double min_time;      /* Minimum time                    */
    double max_time;      /* Maximum time                    */

        /* Variant record of distribution records */
    union {
        struct fixed_rec     fixed;
        struct uniform_rec   uniform;
        struct poisson_rec   poisson;
        struct binomial_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;
    } DIS;
};

    /* Function that returns a (double) value with some distribution
     *  used by source and server to get interarrival times (source)
     *  and service time (server).
     */
double get_time();
```

F.9: stats.h

```
/*****************************************************************
 *  stats.h -- by Edward Vopata
 *****************************************************************
 * Records to be sent the the Statistics Collector by the
 *    Logical Processes &
 * Records for storing the Stats. at the Collector until they can be
 *    sent to the graphics front end (et. al.).
 * Functions for the collection of statistics by the
 *   Logical Processes.
 *      Stats_Init   -- Initialize a stats structure
 *      Stats_Val    -- Add a value to the stats structure.
 *                      Note: must provide for average and STD.
 *      Stats_Mean  --  Return current Average
 *      Stats_STD   --  Return current standard deviation
 *
 * Each stats record contains the ID of the sender, the status of the
 *  stats, and the simulation time of the sender.  The status
 *  indicates whether the stats are Normal (the LP will continue
 *  running), Final (the LP is waiting for a term trans.), or Term
 *  (the LP has exceeded the time specified by sim_term_time and is
 *  signaling the rest of the system that it's termination time).
 *
 * The Collector maintains a singlely linked list of col_rec's,
 *   containing the stats of the logical processes.  These nodes are
 *   ordered by Interval_num (integer) (sim_time / stats_interval).
 *   A set (modified) indicates which logical processes (LP) were
 *   recorded in the node.  The collector uses this set to determine
 *   when to send stats to the graphics front-end.  Not all the stats
 *   that are collected are sent to the graphics front-end.
 * Note: STD == Standard Deviation.  LP = Logical Process(es).
 *****************************************************************
 */

    /* Source Stats. to be sent to the Collecotor */
struct src_stats_rec {
    int      ID;          /* ID of the Source                */
    int      status;      /* Stats. Status of the Source     */
    double   sim_time;    /* Simulation Time of the Source   */
    double   ave_iit;     /* Average Inter-Arrival Time      */
    double   std_iit;     /* STD Inter-Arrival Time          */
    double   max_iit;     /* Max Inter-Arrival Time          */
    long     num_left;    /* Number of item left to be generated.  If
                           * this number is < 0 then it represents the
                           * number of items that have been generated
                           * by an infinite sources.
                           */
};

    /* Queue Stats to be retrieved by the Server */
struct q_stats_rec {
   double   ave_time_in_q;     /* Average Time in the Queue        */
   double   std_time_in_q;     /* STD time in the Queue            */
   double   max_time_in_q;     /* Man time int the Queue           */
```

```
   double   ave_in_q;          /* Ave Number of items in the Queue */
   double   std_in_q;          /* STD Number of items in the Queue */
   double   max_in_q;          /* MAX Number of items in the Queue */
   double   per_full;          /* Percent full                     */
   long     num_in_q;          /* Number of items in the Queue     */
   long     num_through_q;     /* Number of items though the queue */
};

    /* Queue/Server Stats to be sent to the Collector */
struct q_srv_stats_rec {
    int      ID;                  /* ID of the Queue/Server         */
    int      status;              /* Status of the Queue/Server     */
    double   sim_time;            /* Simulation Time of the Server */
    double   per_busy;            /* percent busy (Server)          */
    double   ave_time_in_sys;     /* Average time in the system     */
    double   std_time_in_sys;     /* STD time in the system         */
    double   max_time_in_sys;     /* Max time in the system         */
    double   ave_serve_time;      /* Average Service Time           */
    double   std_serve_time;      /* STD Service Time               */
    double   max_serve_time;      /* Max Service Time               */
    long     num_served;          /* Number of items Serviced       */
    struct   q_stats_rec q_stats; /* Queue Stats record             */
};

    /* Sink Stats to be sent to the Collector */
struct sink_stats_rec {
    int      ID;           /* Id of the Sink                     */
    int      status;       /* Status of the Sink                 */
    double   sim_time;     /* Simulation Time of the Sink        */
    long     num_sunk;     /* Number of items that have been sunk */
};

    /* Statistics Record maintained by the Collector */
struct col_rec {
    int i_num;        /* Interval Number (sim_time/stats_interval)  */
    SET modified;     /* a set to indicate whether stats have been  */
                      /* collected for a particular logical process */
        /* Variant record array of LP statistics */
    union {
        struct src_stats_rec   *Src_stats;   /* Source Stats       */
        struct q_srv_stats_rec *Q_Srv_stats; /* Queue/Server Stats */
        struct sink_stats_rec  *Sink_stats;  /* Sink Stats         */
        char                   *free_stats;  /* Dummy ptr for Free */
    } Stats[MAXPROC];
    struct col_rec *Next;  /* Pointer to next record */
};

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 Structure       */
double Stats_Mean();    /* Return the average of a Stats Structure */
double Stats_STD();     /* Return the STD of a Stats Structure     */
```

F.10: graph.h

```
/********************************************************************
 * graph.h -- by Edward Vopata
 ********************************************************************
 * The Out_Graph:  contains the number of incoming lines, the number
 *  of outgoing lines, and the "id's" of the logical process at the
 *  end of the lines.
 ********************************************************************
 */

struct out_rec {
    int      num_in;         /* Number of incoming lines   */
    int      num_out;        /* Number of outgoing lines   */
    int      out[MAXFAN];    /* id's of lp at end of lines */
};

struct out_list {  /* An out_rec of each lp */
    struct out_rec out_graph[MAXPROC];
};

  /* List used by Resolver processes */
struct list_rec {
    int     ID;                /* ID  of branch process         */
    process Branch Brn_pid;    /* process id of branch          */
    long    state;             /* state of last use             */
    double  sim_time;          /* simulation time of last use   */
    struct  list_rec *Next;    /* point to next record          */
};

  /* Resolver parameter record */
struct res_param_rec {
    int     ID;                   /* Resolver ID                          */
    int     ID2;                  /* Resolver ID secondary                */
    process resolver my_pid;      /* process id of resovler               */
    process deadlock dead;        /* process id of deadlock detector      */
    struct  PS_REC   PS;          /* Structure of all lp process id       */
    struct  out_list out;         /* Out_Graph                            */
    struct  list_rec *list;       /* Branch List                          */
};
```

F.11: rand.h

```
/*******************************************************************
 * 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 pseudo 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               */
# define srand(a) srand48(a)   /* Map srand48 to srand              */
#else /* BSD */
   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))
```

F.12: set.h

```
/*****************************************************************
 * set.h -- by Edward Vopata
 *****************************************************************
 *  Provide bitwise sets operations for large sets.
 *  MAXPROC == Maximum possible number of items in the set.
 *  CHUNKS  == Number of longs (32 bit) needed to satisfy MAXPROC.
 *  Set operations include:
 *    set_clear :  set all bits in the set to zero (0) (uses bzero)
 *    set_add   :  set the corresponding bit to one (1)
 *    set_union :  Or the bits of two (2) sets together
 *    set_in    :  Determine if a bit is set
 *    set_full  :  Determine if all the bits in the set are set.
 *****************************************************************
 */

    /* Determine how many CHUNKS are needed */
#define CHUNKS  ((MAXPROC / 32) + 1)

    /* Number of bits in the set */
#define SETLEN  (CHUNKS * 32)

    /* Short form for a set */
typedef long SET [CHUNKS];

    /* the set_clear function.  Make sure to include the library
     * with bzero. For 3b2's bzero is in -lnet
     */
#define set_clear(set)  bzero(set,sizeof(set))

  /* Forward definition of set functions */
void set_add ();          /* Add an element to a set  */
void set_union();         /* Union of 2 sets          */
int set_in();             /* Is an element in a set?  */
int set_full();           /* Is the set full?         */
```

F.13: mach.h

```
/*********************************************************************
 * 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 compitible machine.
 *           Therefore distribution will be made only on 3b2/3b15's.
 *           ksuvaxl and harris are included for completeness.
 *********************************************************************
 */

#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 */       "ksuvaxl",    /* DEC Vax 11/780 */ /* No D-CCC */
    /* 18 */       "harris",     /* Harris HCX 9   */ /* No D-CCC */
};

/*********************************************************************
 * ksuvaxl & harris were used as development machines.  They are much
 * faster then the 3b2's. ksuvaxl & harris were not compatible 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).
 *********************************************************************
 */
```

F.14: Pid.h

```
/*
 * (c) Copyright 1985 by
 * Computer Technology Research Lab, ATT Bell Laboratories.
 * All rights reserved.
 */
/* Deadlock Detection Mechanism Includes */

#define c_NPROCS    0xffff
#define c_procpid(x)    ((x).u_px < c_nprocs && (x).u_pid == \
            c_procs[(x).u_px].p_pid ? c_procs + (x).u_px : 0)

#define MsgPID(P)    ((c_pid) ((P) << 16))    /* u_seq == 0 */

typedef long c_pid, c_tp;

typedef union {
    c_pid    u_pid;
    c_tp    u_tp;
    struct {

        unsigned short    U_tnum:5,    /* transaction number */
                U_seq:6,    /* c_create() sequence # */
                U_por:5;    /* processor id */
        unsigned short    U_px;        /* proc table index */
    } U;
#define u_px    U.U_px
#define u_por    U.U_por
#define u_seq    U.U_seq
#define u_tnum    U.U_tnum
} c_pidu, c_tpu;
```

F.15: spec.h

```
/* Deadlock Detection Mechanism Includes */

process spec buildgraph(DEADMEN deadmen, Queryserv_tab Query_tab)
{   void trans load_callers (long Proc, CALLERS Callers);
    void trans done();
};

process spec queryserv () {
    void trans putdeadguys (DEADMEN_S Deadmen);
    async trans intercept (c_pidu src_pid, c_pidu dst_pid,
        TPTR_intercept_response response_tptr);
    long trans getstate (c_pidu pid);
};

process spec deadlock (TPTR_report_deadlock report_deadlock,
                    Queryserv_tab_S Query_tab, long D_debug)
{
    trans void putcallers (c_pidu Proc, CALLERS_S Callers);
    async trans intercept (c_pidu src_pid, c_pidu xptr,
        TPTR_intercept_response response_tptr);
    void trans done();
    long trans getstate (c_pidu pid);
    async trans global_detect (DDMSG Ddmsg, c_pidu Newpid);
    async trans term();
    void trans enable_detect();
};

process spec ddagent (c_pidu origin, long seqno) {
    int trans split (c_pidu Pid);
    void trans abort (c_pidu Pid, long Seqno);
    void trans rpt_deadlock (DDMSG DDmsg);
};


/*
 *******************************************************************
 ***  THIS IS THE PROCESS SPECIFICATION FOR THE RESOLVER PROCESS ***
 *******************************************************************
 */
process spec resolver () {
      /* Three asynch transaction to get the resovler its parameters
       *  Because we could not pass structure larger then 4K because
       *  of an bug in the C-Compiler V. 4.1 (sadness).
       */
    async trans setup1 (int,int,process deadlock);
    async trans setup2 (PS_REC);
    async trans setup3 (struct out_list);

       /* part of the intercept_transaction call */
    async trans intercept_response (OUTBUF_S outbuf, int status);
       /* transaction where a deadlock detector report deadlock */
    int  trans report (c_pidu pid);
    async trans term();  /* Terminator */
```

```
};
```

F.16: defs.h

```
/* Deadlock Detection Mechanism Includes */

#define D_NPROCS   50   /* same as NPROCS in libmpcc: nprocs.c */
#define D_NPORS    16   /* same as NPORS in libmpcc: cc.h      */
#define CALLERSIZE  6
#define OUTBUFSIZE 64
#define GDLISTSIZE 50
#define TRUE    1
#define FALSE   0
#define DBG(x)  (d_debug & (1 << (x)))

struct L_LIST {
    c_pidu pid;
    L_LIST *next;
};
typedef struct L_LIST LIST;

struct Dead_ID {
    c_pidu pid;
    long    seqno;
};
typedef struct Dead_ID DEADID;
typedef struct Dead_ID GDLIST [GDLISTSIZE];

struct ddmsg_t {
    process ddagent dda_id;
    long msg_seqno;
    DEADID dlist [GDLISTSIZE];
};
typedef struct ddmsg_t DDMSG;

typedef long CALLERS [CALLERSIZE];
typedef struct {
        CALLERS callers;
    } CALLERS_S;

typedef char OUTBUF [OUTBUFSIZE];
typedef struct {
        OUTBUF outbuf;
    } OUTBUF_S;

typedef trans void (*TPTR_putcallers) (c_pidu, CALLERS_S);
typedef TPTR_putcallers TPTRS_putcallers [D_NPORS];

typedef process deadlock DEADMEN [D_NPORS];
typedef struct {
        DEADMEN deadmen;
    } DEADMEN_S;

typedef process queryserv Queryserv_tab [D_NPORS];
typedef struct {
        Queryserv_tab query_tab;
    } Queryserv_tab_S;
```

```
typedef async trans (*TPTR_intercept_response) (OUTBUF_S, int);
typedef int trans (*TPTR_report_deadlock)(c_pidu);

extern int c_por;

#define ISLOCAL(pid) (pid.u_por == c_por)


#define ACCEPTSTATE(pid) (c_procs[pid.u_px].p_state == c_wselect)
#define WAITSTATE(PID) (c_procs [PID.u_px].p_state == c_wservice || \
            (c_procs [PID.u_px].p_state == c_wmsg &&                \
            c_procs [PID.u_px].p_tccallee != 0))
/*
            c_procs [pid.u_px].p_tcout.tc_callerpid == pid.u_pid))
*/

#define ASSERT(X,Y) \
    if(!(X)) fprintf(stderr,"%s Assertion failed\n",Y)
```

F.17: main.cc

```
#include <signal.h>
#include "dclr.h"

/*********************************************************************
 * main.cc -- by Edward Vopata
 *  Main program of the distributed simulator
 * Usage:
 *  Dsim [-n] [x ...]
 *    where "-n" indicates the use of "stream sockets" where the "n"
 *    is the socket descriptor.
 *    "x" is a Debugging Switch in the range 0..31.
 *    "x"'s may appear in any order.
 *
 * Description:
 *  1. Allocation space of PS (process table) and PS_Data (process
 *     data table).
 *  2. Determine whether "stream sockets" will be used.  This is
 *     determined by a "-n" in the first argument.  The "n" will
 *     be a socket descriptor.  The "-n" is supplied by a start_up
 *     program which opens the socket connection and "exec"'s this
 *     program with a "-n". (See Start.c for description).  This
 *     is done to resolve a naming conflict with Concurrent C and
 *     the socket libraries (the accept statement and the accept
 *     function).  If "stream sockets" are used then the socket
 *     descriptor is store in PS->in_sock and PS->out_sock since
 *     both input and output will go to the same socket, and "Flag"
 *     is set to indicated that "stream sockets" will be used.
 *  3. The Debugging Switches are set.  If "stream sockets" are
 *     used then the first argument is ignored.  The rest of the
 *     arguments indicate which switches to set.  The integer value
 *     of each argument will cause a switch to be set.  A list of
 *     the Debugging Switches and their functions appear at the
 *     bottom of this file.
 *  4. If "stream sockets" are NOT used then the input data will
 *     come from a file.  Prompt the user for the file name and
 *     open the file using (open()) to return a file descriptor.
 *     This will allow the same read and write routines to be
 *     used, since sockets are treated similarly to files.
 *     PS->in_sock gets the file descriptor, PS->out_sock is
 *     set to 1 (stdout) which will redirect the output to the
 *     terminal.
 *  5. Call build_ps() to read the input data and build the PS_Data
 *     table. (see build.cc for description of build_ps()).
 *  6. Call create_ps() to create the processes and build the PS
 *     table.  create_ps will create the processes, handle process
 *     distributions, and process setup. (see create.cc for
 *     description of create_ps()).
 *  7. Deallocate space used by PS and PS_Data, since the processes
 *     have already made a copy of the necessary data by this time.
**********************************************************************
 */

/* Forward Reference for build_ps, create_ps, and print_size */
```

```
void build_ps();
void create_ps();
void print_size();

main(argc,argv)
int  argc;
char **argv;
{
    struct PS_REC *PS;             /* Logical Process Table          */
    union  PS_Data_Rec *PS_Data;   /* Logical Process Data Table     */
    register i;                    /* Index                          */
    int debug_val;                 /* Value of a argv string         */
    int Flag = 0;                  /* Flag to indicated real sockets */
    char name[50];                 /* File name, if not real sockets */

        /* Concurrent C signal handler */
    extern int c_onsig();

        /* Redefine some signals:
         *  Bus Error to just core dump
         *  Illegal Instruction to just core dump
         *  Segmentation Violation to call c_onsig
         */
    signal(SIGBUS,SIG_DFL);
    signal(SIGILL,SIG_DFL);
    (void) signal(SIGSEGV, c_onsig);

        /* Allocate space for the Process Table */
    PS = (struct PS_REC *) malloc (sizeof(struct PS_REC));

        /* Allocate space for the Data Table */
    PS_Data = (union PS_Data_Rec *)
        malloc (sizeof(union PS_Data_Rec) * MAXPROC);

        /* Set All Debugging Switches to  OFF */
    PS->debug = 0;

        /************************************************************
         * Determine if "stream sockets" will be used.  This is done
         * by checking the first argument (argv[1]).  If there is a
         * '-' sign as the first character (argv[1][0]) then "stream
         * sockets" will be used and the value after the "-" is the
         * socket descriptor.
         * Store the socket descriptor in both PS->in_sock  and
         * PS->out_sock since, the socket will be used for both
         * input and output.
         * Set "Flag" to indicate "streams sockets" will be used.
         ************************************************************
         */
    if (argv[1][0] == '-') {
        sscanf(argv[1],"-%d",&PS->in_sock);
        PS->out_sock = PS->in_sock;
        Flag = 1;
    }

        /************************************************************
```

```
     * Determine if any debugging switches are set.  If "Flag" is
     * set then skip to the next argument.  Convert the argument
     * to an integer (into debug_val) and set the corresponding
     * bit in PS->debug.  If the value is "SIZE_OF" then call
     * function "print_size".
     ************************************************************
     */
for (i=1+Flag; i < argc; i++) {
        /* If Debug Mode for Main, print the current argument */
    if (PS->debug&DEBUG_MAIN)
        printf("main:argv[%d] = '%s'\n",i,argv[i]);

        /* Convert string to integer */
    sscanf(argv[i],"%d",&debug_val);

        /* Make sure debug_val is in the proper range befor
         * setting the corresponding bit
         */
    if (0 <= debug_val && debug_val < MAX_DEBUG)
        PS->debug |= 1<<debug_val;

        /* If the SIZE_OF switch is set then call function
         * "print_size" to print the sizes of various
         * structures
         */
    if ((1<<debug_val)==SIZE_OF) { print_size(); continue; }
}

    /************************************************************
     * Handle the case where "stream sockets" are NOT used.
     * In this case, input data will come from a file, so read
     * in the file name, open the file using open (so we can use
     * read and write), and setup the socket (now file)
     * descriptors.
     ************************************************************
     */
if (!Flag) {
        /* Prompt and read the input file name */
    fprintf(stderr,"Enter Input File: ");
    gets(name);
    fprintf(stderr,"\n");

        /* Open a file of input (use file descriptor)
         * This will make the rest of the program think we
         * are using sockets.  PS->in_sock will be used for
         * input.
         */
    if ((PS->in_sock = open(name,0)) < 0) {
        fprintf(stderr,"main: File '%s' does Not Exist\n",name);
        c_exit(0);
    };
        /* PS->out_sock will be used for output. So send the
         * output to "stdout" (the terminal).
         */
    PS->out_sock = 1;
}
```

```
        /* Build the PS_Data Table */
    build_ps(PS,PS_Data);

        /* Create the Logical process and
         * build the Process Table.
         */
    create_ps(PS,PS_Data);

        /* Clean up:  Free allocated space for PS and PS_Data */
    for (i=0; i < PS->num_nodes; i++)
        free((char *) PS_Data[i].Src);
    free((char *)PS);
    free((char *)PS_Data);
}

/*********************************************************************
 * Debugging Switches Identification and Functions
 *
 * Switch
 *Number  Function
 *------  --------
 *  0     Debug Mode for main.cc     (main program)
 *  1     Debug Mode for build.cc    (read input data)
 *  2     Debug Mode for create.cc   (create logical processes)
 *  3     Debug Mode for source.cc   (source logical process)
 *  4     Debug Mode for sink.cc     (sink logical process)
 *  5     Debug Mode for server.cc   (server process)
 *  6     Debug Mode for queue.cc    (queue process)
 *  7     Debug Mode for branch.cc   (branch logical process)
 *  8     Debug Mode for col.cc      (stats. collector)
 *  9     Debug Mode for resolver.cc (deadlock resolver)
 * 10     Debug Mode for distrib.cc  (stochastic distribution func.)
 * 11     Debug Mode for term.cc     (terminator process)
 * 12     Debug Mode for sock.cc     (socket communication)
 * 13     Debug Mode for start.c     (socket connection)
 * 14     Print Report of collected statistics to file "Stats_Report"
 * 15     Print Size_of Report of sizes of various structures to
 *           file "Size_of"
 * 16     Print Out_Graph Report of graph and pid's of simulation
 *         models to file "Out_Graph"
 * 17-19   Unused
 * 20-31   Used by Scott Hammond's Deadlock Detection Program.
 *********************************************************************
 */
```

F.18: build.cc

```
#include "dclr.h"

/*****************************************************************
 * build.cc  -- By Edward Vopata
 *  Read the input model, parse the input model, create the process
 *  table, and process data table.
 *****************************************************************
 */

/*****************************************************************
 * Function:
 *   build_dis()
 * Parameter:
 *    ptr   - point into the input model string
 *    dis   - distribution record
 *    debug - debug flag.
 * Summary:
 *   Parse the input model for the stochastic distribution functions
 *   and their parameters and store the information in "dis"
 *****************************************************************
 */
void build_dis();

static void build_dis(ptr,dis,debug)
char   *ptr;
struct distrib_rec *dis;
int    debug;
{
        /* Get the function time, min, and max times */
    sscanf(ptr,"%d %lf %lf", &dis->type, &dis->min_time,
        &dis->max_time);
    if (debug)  /* Distrib tracing */
      printf("build_dis: type %d min %lf  max %lf\n",
        dis->type,dis->max_time,dis->max_time);

    dis->debug = debug;

       /* Determine which type of function, and get the
        * parameters from the input model.
        */
    switch (dis->type) {
    case FIXED:
        sscanf(ptr,"%*d %*lf %*lf %lf", &dis->DIS.fixed.time);
    if (debug)
        printf("build_dis:Fixed: %lf\n",dis->DIS.fixed.time);
        break;

    case UNIFORM:
        sscanf(ptr,"%*d %*lf %*lf %ld %ld",
            &dis->DIS.uniform.lower, &dis->DIS.uniform.upper);
        break;

    case POISSON:
```

```
        sscanf(ptr,"%*d %*lf %*lf %lf", &dis->DIS.poisson.mean);
    if (debug)
        printf("build_dis:POISSON: %lf\n",dis->DIS.poisson.mean);
        break;

    case BINOMIAL:
        sscanf(ptr,"%*d %*lf %*lf %ld %lf",
            &dis->DIS.binomial.trials, &dis->DIS.binomial.prob);
        break;

    case EXPNTL:
        sscanf(ptr,"%*d %*lf %*lf %lf", &dis->DIS.expntl.mean);
    if (debug)
        printf("build_dis:EXPNTL: %lf\n",dis->DIS.expntl.mean);
        break;

    case NORMAL:
        sscanf(ptr,"%*d %*lf %*lf %lf %lf",
            &dis->DIS.normal.mean, &dis->DIS.normal.stdev);
        break;

    case GAMMA:
        sscanf(ptr,"%*d %*lf %*lf %lf %lf",
            &dis->DIS.gamma.mean, &dis->DIS.gamma.k);
        break;

    case BETA:
        sscanf(ptr,"%*d %*lf %*lf %lf %lf",
            &dis->DIS.beta.k1, &dis->DIS.beta.k2);
        break;

    case ERLANG:
        sscanf(ptr,"%*d %*lf %*lf %lf %ld",
            &dis->DIS.erlang.mean, &dis->DIS.erlang.k);
        break;

    case LOGNORMAL:
        sscanf(ptr,"%*d %*lf %*lf %lf %lf",
            &dis->DIS.lognormal.mean, &dis->DIS.lognormal.stdev);
        break;

    case WEIBULL:
        sscanf(ptr,"%*d %*lf %*lf %lf %lf",
            &dis->DIS.weibull.shape, &dis->DIS.weibull.scale);
        break;

    default:
        dis->type = FIXED;
        dis->min_time = 0;
        dis->max_time = 0;
        dis->DIS.fixed.time = 1;
        break;
    }
}
```

```
/*******************************************************************
 * Function:
 *   build_ps()
 * Parameter:
 *   PS      -  Process table
 *   PS_Data -  Process data table
 * Summary:
 *   Read and parse the input model, store information in the
 *   process table and the process data table.
 *   The input model is discussed in Appendix C.
 *******************************************************************
 */
void build_ps(PS,PS_Data)
struct PS_REC *PS;
union PS_Data_Rec *PS_Data;
{
    int  type,id;          /* Indexes                              */
    int  Done = FALSE;     /* Flag                                 */
    register  i, fan_num;  /* Indexes                              */
    char *ptr;             /* pointer into the line                */
    char line[MAXLINE+1];  /* Incoming line of the input model */

        /* Initialize some variables */
    PS->num_nodes      = 0;
    PS->Total_Gen      = 0;
    PS->Max_Hop_Count = 0;
    PS->Infinite_Src   = FALSE;

    while(!Done) {
            /* get a line from the socket   */
            /* line = "( ( type id ) ????)" */
        getline(PS->in_sock,line,PS->debug&DEBUG_SOCK);
            /* Ignore lines that begin with a "#" or a "\n" */
        if (line[0] == '#') continue;  /* # in column 1     */
        if (line[0] == '/n') continue; /* True blank line   */

            /* point to "( type id ) ????" */

        ptr = index(line,'(') + 1;
        ptr = index(ptr,'(') + 1;
           /* Get the type and id of the logical process */
        sscanf(ptr,"%d %d",&type,&id);

        if (PS->debug&DEBUG_BUILD)  /* Build tracing */
            printf("Build: NewLine = [%s]\n",ptr);

        if (PS->debug&DEBUG_BUILD) /* Build tracing */
            printf("Build: (type id) = (%d %d)\n",type,id);

        ptr = index(ptr,')') + 1;

        if (PS->debug&DEBUG_BUILD)  /* Build tracing */
            printf("Build: NewLine = [%s]\n",ptr);

           /* Determine the type of logical process, create a
            * entry in the PS_Data table, and parse the input
```

```
     * model and store the information in the PS_Data
     * table.
     */
switch (type)
{
case SOURCE:  /* For Source LP's */
     PS->num_nodes += 1;
     PS->type[id] = SOURCE;
     PS_Data[id].Src =
         (struct Src_rec *)malloc(sizeof(struct Src_rec));

     ptr = index(ptr,'(') + 1;
     build_dis(ptr, &PS_Data[id].Src->dis,
         PS->debug&DEBUG_DISTRIB);
     ptr = index(ptr,')') + 1;

        /* "iit servt ngen fanout (( x x) (x x) ..)" */
     sscanf(ptr,"%d %d %ld %d",
         &PS->mach[id],
         &PS->virt[id],
         &PS_Data[id].Src->num_Gen,
         &PS_Data[id].Src->Out_ID);

     if (PS->debug&DEBUG_BUILD) {
       printf("Build:Src:(%d %d) mach %d virt %d ",
          type, id, PS->mach[id],PS->virt[id]);
       printf("nGen %ld Out_ID %d\n",
          PS_Data[id].Src->num_Gen,
          PS_Data[id].Src->Out_ID);
     }

        /* Is the source an infinite one? */
     PS->Total_Gen += PS_Data[id].Src->num_Gen;
     if (PS_Data[id].Src->num_Gen == 0)
         PS->Infinite_Src = TRUE;
     break;

case SINK:  /* For Sink LP's */

     PS->num_nodes += 1;
     PS->type[id] = SINK;
     PS_Data[id].Sink =
         (struct Sink_rec *)malloc(sizeof(struct Sink_rec));

     sscanf(ptr,"%d %d %d",
         &PS->mach[id],
         &PS->virt[id],
         &PS_Data[id].Sink->Num_fan_in);

     if (PS->debug&DEBUG_BUILD) {
       printf("Build:Sink: (%d,%d) mach %d virt %d ",
          type,id,PS->mach[id],PS->virt[id]);
       printf("Num_fan_in %d\n",
          PS_Data[id].Sink->Num_fan_in);
     }
     break;
```

```
case QUE_SRV:  /* The Queue & Server LP's */
     PS->num_nodes += 1;
     PS->type[id]   = QUE_SRV;
     PS->Max_Hop_Count += 1;
     PS_Data[id].Q_Srv =
         (struct Q_Srv_rec *)malloc(sizeof(struct Q_Srv_rec));

ptr = index(ptr,'(') + 1;
     build_dis(ptr, &PS_Data[id].Q_Srv->dis,
          PS->debug&DEBUG_DISTRIB);
     ptr = index(ptr,')') + 1;

     sscanf(ptr,"%d %d %d %d %d %d",
          &PS->mach[id],
          &PS->virt[id],
          &PS_Data[id].Q_Srv->Out_ID,
          &PS_Data[id].Q_Srv->Q_size,
          &PS_Data[id].Q_Srv->Q_method,
          &PS_Data[id].Q_Srv->Num_fan_in);

     if (PS->debug&DEBUG_BUILD) {
          printf("Build:Q_Srv:(%d %d) mach %d virt %d ",
               type,id, PS->mach[id], PS->virt[id]);
          printf("Out ID %d Qsize %d Qmeth %d #in %d\n",
               PS_Data[id].Q_Srv->Out_ID,
               PS_Data[id].Q_Srv->Q_size,
               PS_Data[id].Q_Srv->Q_method,
               PS_Data[id].Q_Srv->Num_fan_in);
     }
     break;

case BRANCH:  /* The branch LP's */
     PS->num_nodes += 1;
     PS->type[id] = BRANCH;
     PS_Data[id].Branch =
        (struct Branch_rec *)malloc(sizeof(struct Branch_rec));
     sscanf(ptr,"%d %d %d %d",
          &PS->mach[id],
          &PS->virt[id],
          &PS_Data[id].Branch->Num_fan_in,
          &PS_Data[id].Branch->Num_fan_out);
     ptr = index(ptr,'(') + 1;

     if (PS->debug&DEBUG_BUILD) {
          printf("Build:Branch (%d %d): mach %d virt %d ",
               type,id,PS->mach[id], PS->virt[id]);
          printf("#In %d #Out %d\n",
               PS_Data[id].Branch->Num_fan_in,
               PS_Data[id].Branch->Num_fan_out);
     }

     fan_num = PS_Data[id].Branch->Num_fan_out;

     for (i=0; i < fan_num; i++) {
          ptr = index(ptr,'(') + 1;
          sscanf(ptr,"%d %lf",
```

```
                    &PS_Data[id].Branch->Fan_out[i].ID,
                    &PS_Data[id].Branch->Fan_out[i].prob);

                if (PS->debug&DEBUG_BUILD)
                  printf("Build:Branch %d: FanOut:ID %d prob: %lf\n",
                    id,
                    PS_Data[id].Branch->Fan_out[i].ID,
                    PS_Data[id].Branch->Fan_out[i].prob);

                    ptr = index(ptr,')') + 1;
                    if (PS->debug&DEBUG_BUILD)
                        printf("Build: NewLine = [%s]\n",ptr);
            }
            break;

          /* Indicates End of Input Model */
        case SIG_ENDS:  /* "((99 1))  message */
                /* Wait for the "SIG_ENDS */
            if (PS->debug&DEBUG_BUILD)
                printf("SIGNAL: (%d %d)\n",type,id);
            break;

          /* Indicates the Start of the simulation */
        case SIG_SOS:
               /* Get Termination Time and Stats interval */
            sscanf(ptr,"%lf %lf",
                &PS->sim_term_time, &PS->stats_interval);

            if (PS->debug&DEBUG_BUILD)
                printf("sim_term_time %lf stats_interval %lf\n",
                    PS->sim_term_time, PS->stats_interval);
            Done = TRUE;
            break;

        default:
            fprintf(stderr,"Build: Unknown type (%d)\n",type);
        }
    }
    PS->Max_Hop_Count += 1;  /* Deal with HOP counter */
}
```

F.19: create.cc

```
#include "dclr.h"
#include "graph.h"
#include "Pid.h"
#include "defs.h"
#include "spec.h"
#include "mach.h"

/*********************************************************************
 * create.cc -- by Edward Vopata
 *********************************************************************
 *  Create and distribute the logical processes, create the
 *  deadlock detector and resolver processes.  Send every process
 *  its parameter table
 *********************************************************************
 */
SEND get_send_tp();

/*********************************************************************
 * Function:
 *   get_send_tp()
 * Parameter:
 *   PS - process table
 *   ID - logical process ID
 * Summary:
 *   return a transaction pointer to the logical process ID.
 *********************************************************************
 */
static SEND get_send_tp(PS,ID)
struct PS_REC *PS;
int    ID;
{

      /* Determine the type of logical process and
       * return a transaction pointer to it
       */
    switch(PS->type[ID]) {
        case QUE_SRV:
            return PS->Que_pid[ID].send;
        case BRANCH:
            return PS->ps_pid[ID].Branch_pid.send;
        case SINK:
            return PS->ps_pid[ID].Sink_pid.send;
        default:
            fprintf(stderr,"get_send_tp: Invalid Type (%d)\n",
                PS->type[ID]);
            return NULL;
    }
}


/*********************************************************************
 * Function:
 *   create_ps()
```

```
 * Parameter:
 *   PS      - Process table
 *   PS_Data - Process data table
 * Summary:
 *    1. Create & distribute the logical processes, record pid in PS
 *    2. Create Collector & Terminator process
 *    3. Build an Out Graph
 *    4. Create & distribute deadlock detectors and resolvers
 *    5. Set up deadlock detectors and resolvers
 *    6. Set up the other logical processes.
 ******************************************************************
 */
void create_ps (PS,PS_Data)
struct PS_REC *PS;
union  PS_Data_Rec *PS_Data;
{
    register i,j,id;           /* Indexes               */
    int    num_VP = 0;         /* Number of processors */
#ifdef SYS5_DIS
    int    Mach_VP[MAX_MACH]; /* list of machines     */
    int    VP[D_NPORS];        /* list of processors   */
#endif
    union param_rec {       /* variant record of parameter tables */
        struct src_param  src;
        struct sink_param sink;
        struct srv_param  srv;
        struct q_param    que;
        struct brn_param  brn;
        struct col_param  col;
        struct term_param term;
    } Param;
       /* Transaction pointers to the collector */
    trans void (*src_stats_tp)(struct src_stats);
    trans void (*srv_stats_tp)(struct srv_stats);
    trans void (*snk_stats_tp)(struct snk_stats);

    struct out_list  *out;      /* Out graph */

       /* Deadlock Detection definitions */
    Queryserv_tab_S  Qs_tab;
    DEADMEN_S   deadmen;
    CALLERS     caller;
    process     buildgraph Build_graph;

    process     resolver   Res[MAX_MACH];  /* list of resolvers   */
    char        hostname[10];         /* name of the hosts machine */
    int         hostid;               /* id of the hosts machine   */

       /* get the name of the host machine, the machine that the
      . * distributed simulator started on.
        */
    gethostname(hostname,10);

        /* Find the id of the host */
    for (i=0;i < MAX_MACH; i++)
        if (strcmp(hostname,mach_name[i]) == 0) hostid = i;
```

```
    /* Start any other processors as needed */
#ifdef SYS5_DIS
    bzero(Mach_VP,sizeof(Mach_VP));
    for (i=0; i < PS->num_nodes; i++) {
        if (Mach_VP[PS->mach[i]] == 0 && PS->mach[i] != hostid)  {
            Mach_VP[PS->mach[i]] =
                c_processor(mach_name[PS->mach[i]],LOCATE);
        }
    }
#endif

     /* For each logical process in the process table (PS)
      * create the logical process on the specified processor
      */
    for (i=0; i < PS->num_nodes; i++) {
        switch(PS->type[i]) {
        case SOURCE:   /* The Source LP */
            PS->ps_pid[i].Src_pid = create Source ()
#ifdef SYS5_DIS
                processor(Mach_VP[PS->mach[i]])
#endif
            ;

            if (PS->debug&DEBUG_CREATE) /* Create tracing */
                printf("Create: Source Created on machine %s\n",
                    mach_name[PS->mach[i]]);
            break;

        case SINK:  /* The Sink LP */
            PS->ps_pid[i].Sink_pid = create Sink ()
#ifdef SYS5_DIS
                processor(Mach_VP[PS->mach[i]])
#endif
            ;

            if (PS->debug&DEBUG_CREATE) /* Create tracing */
                printf("Create: Sink Created on machine %s\n",
                    mach_name[PS->mach[i]]);
            break;


        case QUE_SRV: /* The Queue and Server LP */
            PS->ps_pid[i].Srv_pid = create Server ()
#ifdef SYS5_DIS
                processor(Mach_VP[PS->mach[i]])
#endif
            ;


            PS->Que_pid[i] = create Queue ()
#ifdef SYS5_DIS
                processor(Mach_VP[PS->mach[i]])
#endif
            ;

            if (PS->debug&DEBUG_CREATE) /* Create tracing */
```

```
                printf("Create: Queue/Server created on machine %s\n",
                    mach_name[PS->mach[i]]);
            break;


        case BRANCH: /* The Branch LP */
            PS->ps_pid[i].Branch_pid = create Branch ()
#ifdef SYS5_DIS
                processor(Mach_VP[PS->mach[i]])
#endif
            ;

            if (PS->debug&DEBUG_CREATE) /* Create tracing */
               printf("Create: Branch created on machine %s\n",
                   mach_name[PS->mach[i]]);
            break;


        default:
            fprintf(stderr,"get_send_tp: Invalid Type (%d)\n",
                PS->type[i]);
            break;
        }
    }

      /* Create the Collector and Terminator */
    PS->Col_pid  = create Collector ();
    PS->Term_pid = create Terminator ();

       /* Build an Out Graph.  A list of the process to whom
        * a logical process can send messages
        */
    out = (struct out_list *)malloc(sizeof(struct out_list));
    bzero(out,sizeof(struct out_list));
    bzero(deadmen.deadmen,sizeof(DEADMEN_S));
    bzero(Qs_tab.query_tab,sizeof(Queryserv_tab_S));

    for (i=0; i < PS->num_nodes; i++) {
       switch(PS->type[i]) {
       case SOURCE:
         out->out_graph[i].num_in = 0;
         out->out_graph[i].num_out = 1;
         out->out_graph[i].out[0] = PS_Data[i].Src->Out_ID;
         break;

       case SINK:
         out->out_graph[i].num_in = PS_Data[i].Sink->Num_fan_in;
         out->out_graph[i].num_out = 0;
         break;

       case QUE_SRV:
         out->out_graph[i].num_in = PS_Data[i].Q_Srv->Num_fan_in;
         out->out_graph[i].num_out = 1;
         out->out_graph[i].out[0] = PS_Data[i].Q_Srv->Out_ID;
         break;
```

```
        case BRANCH:
          out->out_graph[i].num_in = PS_Data[i].Branch->Num_fan_in;
          out->out_graph[i].num_out = PS_Data[i].Branch->Num_fan_out;
          for (j=0; j < out->out_graph[i].num_out; j++) {
              out->out_graph[i].out[j] =
                 PS_Data[i].Branch->Fan_out[j].ID;
          }
          break;

        default:  fprintf(stderr,"Invalid type (%d)\n",PS->type[i]);
        }
    }


/* Create the Deadlock Detector processes */
    bzero(VP,sizeof(VP));
    VP[0] = 0;
    for (i=0,num_VP=1; i  < MAX_MACH; i++)
        if (Mach_VP[i] != 0) VP[num_VP++] = Mach_VP[i];

    for (i=0; i < num_VP; i++) {
        Res[i] = create resolver()  /* Create a resolver */
#ifdef SYS5_DIS
            processor(VP[i]);
#endif
        ;
        c_setpriority(Res[i],-15);

        Qs_tab.query_tab[VP[i]] = create queryserv()
#ifdef SYS5_DIS
            processor(VP[i]);
#endif
        ;
    }

    for (i=0; i < num_VP; i++) {

        deadmen.deadmen[VP[i]] =
            create deadlock(Res[i].report,Qs_tab,PS->debug)
#ifdef SYS5_DIS
            processor(VP[i])
#endif
        ;
        c_setpriority(deadmen.deadmen[VP[i]],-20);
    }

    Build_graph= create buildgraph(deadmen.deadmen,Qs_tab.query_tab);

       /* The the deadlock detection mechanism about the graph of
        * the simulation (Out Graph)
        */
    for (i=0; i < PS->num_nodes; i++) {
        if (out->out_graph[i].num_out == 0) continue;
        bzero(caller,sizeof(CALLERS));
        for (j=0; j < out->out_graph[i].num_out; j++) {
            id = out->out_graph[i].out[j];
```

```
        if (PS->type[id] == QUE_SRV)
            caller[j] = (long)PS->Que_pid[id];
        else   caller[j] = PS->ps_pid[id].pid;
    }
    if (PS->type[i] == QUE_SRV) {
        caller[j] = (long) PS->Que_pid[i];
    }
    Build_graph.load_callers(PS->ps_pid[i].pid,caller);
}
Build_graph.done();

  /* Send the resolver its parameter table */
for (i=0; i < num_VP; i++) {
    Res[i].setup1(RES_ID,i,deadmen.deadmen[VP[i]]);
    Res[i].setup2(*PS);
    Res[i].setup3(*out);
}

   /* create transaction pointers to the collector */
src_stats_tp = PS->Col_pid.Src_stats;
srv_stats_tp = PS->Col_pid.Que_Srv_stats;
snk_stats_tp = PS->Col_pid.Sink_stats;


    /* For each logical process in the process table
     * send the LP its parameter table.
     */
for (i=0; i < PS->num_nodes; i++) {
    switch(PS->type[i]) {
    case SOURCE:  /* The Source LP */
        if (PS->debug&DEBUG_CREATE) /* Create tracing */
            printf("Create: Source setup\n");
        Param.src.ID              = i;
        Param.src.sim_term_time  = PS->sim_term_time;
        Param.src.stats_interval = PS->stats_interval;
        Param.src.Max_Hop_Count  = PS->Max_Hop_Count;
        Param.src.num_Gen = PS_Data[i].Src->num_Gen;
        Param.src.Out_ID  = PS_Data[i].Src->Out_ID;
        Param.src.send    = get_send_tp(PS,Param.src.Out_ID);
        Param.src.stats   = src_stats_tp;
        Param.src.dis     = PS_Data[i].Src->dis;
        Param.src.debug   = PS->debug;
        PS->ps_pid[i].Src_pid.setup(Param.src);
        break;

    case SINK:   /* The Sink LP */
        if (PS->debug&DEBUG_CREATE) /* Create tracing */
            printf("Create: Sink setup\n");
        Param.sink.ID              = i;
        Param.srv.sim_term_time    = PS->sim_term_time;
        Param.sink.stats_interval = PS->stats_interval;
        Param.sink.Num_Fan_In = PS_Data[i].Sink->Num_fan_in;
        Param.sink.stats        = snk_stats_tp;
        Param.sink.debug        = PS->debug;
        PS->ps_pid[i].Sink_pid.setup(Param.sink);
        break;
```

```
        case QUE_SRV:  /* The Queue & Server LP */
            if (PS->debug&DEBUG_CREATE) /* Create tracing */
                printf("Create: Queue/Server setup\n");
            Param.que.ID            = i;
            Param.que.Q_size        = PS_Data[i].Q_Srv->Q_size;
            Param.que.Q_method      = PS_Data[i].Q_Srv->Q_method;
            Param.que.Num_Fan_In    = PS_Data[i].Q_Srv->Num_fan_in;
            Param.que.debug         = PS->debug;
            PS->Que_pid[i].setup(Param.que);

            Param.srv.ID                = i;
            Param.srv.sim_term_time     = PS->sim_term_time;
            Param.srv.stats_interval    = PS->stats_interval;
            Param.srv.Max_Hop_Count     = PS->Max_Hop_Count;
            Param.srv.Out_ID = PS_Data[i].Q_Srv->Out_ID;
            Param.srv.send    = get_send_tp(PS,Param.srv.Out_ID);
            Param.srv.stats   = srv_stats_tp;
            Param.srv.Que     = PS->Que_pid[i];
            Param.srv.dis     = PS_Data[i].Q_Srv->dis;
            Param.srv.debug   = PS->debug;
            PS->ps_pid[i].Srv_pid.setup(Param.srv);
            break;

        case BRANCH:  /* The Branch LP */
            if (PS->debug&DEBUG_CREATE) /* Create tracing */
                printf("Create: Branch setup\n");
            Param.brn.ID                = i;
            Param.brn.sim_term_time     = PS->sim_term_time;
            Param.brn.stats_interval    = PS->stats_interval;
            Param.brn.Max_Hop_Count     = PS->Max_Hop_Count;
            Param.brn.Num_Fan_In   =PS_Data[i].Branch->Num_fan_in;
            Param.brn.Num_Fan_Out  =
                PS_Data[i].Branch->Num_fan_out;

            for (j=0; j < Param.brn.Num_Fan_Out; j++) {
              Param.brn.Fan_Out[j].ID =
                PS_Data[i].Branch->Fan_out[j].ID;
              Param.brn.Fan_Out[j].prob =
                PS_Data[i].Branch->Fan_out[j].prob;
              Param.brn.Fan_Out[j].send=
                get_send_tp(PS,PS_Data[i].Branch->Fan_out[j].ID);
            }
            Param.brn.debug = PS->debug;
            PS->ps_pid[i].Branch_pid.setup(Param.brn);
            break;

        default:
           fprintf(stderr,"Create:Invalid type (%d)\n", PS->type[i]);
           break;
        }

    }

        /* Send the Collector's parameter table */
    Param.col.ID = COL_ID;
    Param.col.PS = *PS;
```

```
    PS->Col_pid.setup(Param.col);

       /* Send the Terminator's parameter table */
    Param.term.ID = TERM_ID;
    Param.term.PS = *PS;
    PS->Term_pid.setup(Param.term);
    PS->Term_pid.setup1(num_VP,Res);

       /* If Switch 16 is set, create an Out Graph report */
    if (PS->debug&OUT_GRAPH) /* OUT GRAPH */
        print_out_graph(PS,out,Res,num_VP);

    free(out);  /* Clean up */
}
```

F.20: source.cc

```
#include "dclr.h"

/*******************************************************************
 * source.cc -- by Edward Vopata
 *******************************************************************
 *  The Source Process.
 *  This process generates messages at a specified arrival rate and
 *  sends the message to a specified logical process.  The source
 *  represent a point in the physical system where jobs enter the
 *  system.
 *******************************************************************
 */
process body Source ()
{
    process  Source Iam;                   /* Source's Process id      */
    ITEM     Item;                         /* A message                */
    STATS    iit;                          /* inter-arrival time stats */
    struct   src_stats_rec Src_stats;      /* Source Statistics        */
    struct   src_param Param;              /* Source's Parameter table */
    double   arrive_time;                  /* arrival time             */
    long     i_num = 0;                    /* Interval number          */


       /* Seed the random number generator */
    srand(getpid() * getppid());

       /* Get the Source's Process id */
    Iam = (process Source)c_mypid();

       /* Get the Source's Parameter table */
    accept setup(Src) {Param = Src; };

       /* Initialize the Statistics */
    Stats_Init(&iit);
    Src_stats.ID       = Param.ID;
    Src_stats.status   = STATS_NORMAL;
    Src_stats.sim_time = 0.0;


    for (;;) {
          /* get an arrival time using a distribution function */
        arrive_time = get_time(Param.dis);

          /* Has the simulation termination time been exceeded? */
        if (Param.sim_term_time > 0 &&
          Src_stats.sim_time + arrive_time > Param.sim_term_time) {
            Src_stats.status = STATS_TERM;
            break;
        }

          /* Has a stats interval occurred? */
        if (i_num != (long)((Src_stats.sim_time + arrive_time) /
          Param.stats_interval)) {
```

```
            /* If so, calculate a new i_num */
          i_num = (long)((Src_stats.sim_time + arrive_time) /
              Param.stats_interval);
            /* Calculate new stats */
          Src_stats.ave_iit  = Stats_Mean(&iit);
          Src_stats.std_iit  = Stats_STD(&iit);
          Src_stats.max_iit  = iit.max_val;
          Src_stats.num_left = Param.num_Gen - iit.num_val;

            /* Send the stats to the collector */
          (*Param.stats)(Src_stats);
          if (Param.debug&DEBUG_SOURCE) /* Source tracing */
              printf("Source %d: Sending Stats\n",Param.ID);
      }
         /* Record stats for current message */
      Stats_Val(&iit,arrive_time);
      Src_stats.sim_time += arrive_time;
      if (Param.debug&DEBUG_SOURCE) /* Source tracing */
          printf("Source %d: Arrival Time %lf\n",
              Param.ID,arrive_time);

        /* Generate a new message */
      Item.Time_Stamp   = Src_stats.sim_time;  /* Timestamp      */
      Item.Arrive_Time  = Src_stats.sim_time;  /* Start Time     */
      Item.Priority     = rand() % MAXPRIOR;   /* Priority       */
      Item.Item_type    = MSG_ITEM;            /* Normal Message */
      Item.Id_of_Src    = Param.ID;            /* Id of source   */
      Item.data[0]      = '\0';                /* NO extra data  */

         /* Send the message */
      (*Param.send)(Item);
      if (Param.debug&DEBUG_SOURCE)  /* Source tracing */
          printf("Source %d: Sending Item %lf\n",
              Param.ID,Item.Time_Stamp);

        /* Has a terminate message arrive? or
         * Has the source generated the specified number of
         * messages?
         */
      if (c_transcount(Iam.term) ||
         ((Param.num_Gen > 0) && (iit.num_val >= Param.num_Gen))) {
          Src_stats.status = STATS_FINAL;
          break;
      }
  }

  if (Param.debug&DEBUG_SOURCE) /* Source tracing */
      printf("Source %d: Sending Final Stats\n",Param.ID);

     /* Calculate Final Statistics */
  Src_stats.ave_iit  = Stats_Mean(&iit);
  Src_stats.std_iit  = Stats_STD(&iit);
  Src_stats.max_iit  = iit.max_val;
  Src_stats.num_left = Param.num_Gen - iit.num_val;
  (*Param.stats)(Src_stats);  /* Send the Final Stats */
```

```
  /* Send Null_Msg with a VERY large Time Stamp */
Item.Time_Stamp    = MAX_TIME;   /* VERY LARGE TIMESTAMP */
Item.Arrive_Time   = Src_stats.sim_time;
Item.Priority      = 0;
Item.Item_type     = MSG_NULL;
Item.Id_of_Src     = Param.ID;
Item.data[0]       = '\0';
(*Param.send)(Item);  /* Send the NULL message */

accept term();  /* Accept the terminate message */

if (Param.debug&DEBUG_SOURCE) /* Source tracing */
    printf("Source %d:  Terminated\n",Param.ID);

/* Terminate */
}
```

F.21: server.cc

```
#include "dclr.h"

/*****************************************************************
 * server.cc -- by Edward Vopata
 *****************************************************************
 * The Server Process:
 *  The server is a station in the physical system.  The server
 *  services jobs and sends them to the next logical process.  The
 *  server gets it jobs from its associated queue.
 *****************************************************************
 */
process body Server()
{
    process Server Iam;                  /* Server's process id      */
    struct  srv_param Param;             /* Server's parameter table */
    struct  q_srv_stats_rec Srv_stats;   /* Server's Statistics      */
    STATS   sys_time;                    /* system time stats        */
    STATS   serve_stats;                 /* service time stats       */
    ITEM    Item;                        /* message buffer           */
    double  serve_time;                  /* service time             */
    double  sum_serve_time = 0.0;        /* sum of the service times */
    long    i_num    = 0;                /* interval number          */

      /* Seed the random number generator */
    srand(getpid() * getppid());

      /* Get the Server's Process id */
    Iam = (process Server)c_mypid();

      /* Get the Server's Parameter table */
    accept setup(Srv) {Param = Srv; };

      /* Initialize the server's statistics table */
    Srv_stats.ID          = Param.ID;
    Srv_stats.status      = STATS_NORMAL;
    Srv_stats.sim_time    = 0.0;
    Srv_stats.num_served = 0;
    Stats_Init(&sys_time);
    Stats_Init(&serve_time);

    for (;;) {

            /* Get a message for the server's associated queue */
        Param.Que.get_item(&Item,Srv_stats.sim_time);

        if (Param.debug&DEBUG_SERVER)  /* Server tracing */
            printf("Srv %d: Get_item time %lf\n",
                Param.ID, Item.Time_Stamp);

            /* Is this a normal messages? */
        if (Item.Item_type != MSG_ERR) {

                /* Generate a service time */
```

```
 serve_time = get_time(Param.dis);

 if (Param.debug&DEBUG_SERVER) /* Server tracing */
     printf("Srv %d: serve_time %lf\n",
          Param.ID, serve_time);

    /* Has simulation termination time been exceeded? */
if (Param.sim_term_time > 0 &&
    Srv_stats.sim_time+serve_time > Param.sim_term_time)
{
    Srv_stats.status    = STATS_TERM;
    break;
}

  /* Has an interval occurred? */
if (i_num != (long)(MAX(Srv_stats.sim_time + serve_time,
   Item.Time_Stamp + serve_time) / Param.stats_interval))
{
      /* If so, update i_num */
   i_num = (long)(MAX(Srv_stats.sim_time + serve_time,
   Item.Time_Stamp + serve_time) / Param.stats_interval);

      /* Calculate server statistics */
   Srv_stats.ave_time_in_sys  = Stats_Mean(&sys_time);
   Srv_stats.std_time_in_sys  = Stats_STD(&sys_time);
   Srv_stats.max_time_in_sys  = sys_time.max_val;
   Srv_stats.ave_serve_time   = Stats_Mean(&serve_stats);
   Srv_stats.std_serve_time   = Stats_STD(&serve_stats);
   Srv_stats.max_serve_time   = serve_stats.max_val;
   Srv_stats.per_busy  = (Srv_stats.sim_time) ?
     (double)(100 * sum_serve_time)/Srv_stats.sim_time :
      0.0;
   Srv_stats.q_stats =
        Param.Que.stats(Srv_stats.sim_time);
      /* Send stats to collector */
   (*Param.stats)(Srv_stats);
}

  /* Update simulation clock and the message's timestamp
   * Clock = MAX (timestamp,Clock) + service time
   */
Srv_stats.sim_time = Item.Time_Stamp =
    MAX(Srv_stats.sim_time + serve_time,
         Item.Time_Stamp + serve_time);

  /* Only record stats for normal messages, ignore
   * NULL messages.
   */
if (Item.Item_type == MSG_ITEM) {
    sum_serve_time += serve_time;
    Srv_stats.num_served += 1;
    Stats_Val(&serve_stats,serve_time);
    Stats_Val(&sys_time,
         Srv_stats.sim_time - Item.Arrive_Time);
}
else {
```

```
                /* If the message is a NULL message, then
                 * decrement the NULL's hop counter
                 */
            Item.Hop_Count -= 1;
        }

            /* Send a normal message, or a NULL with a hop
            * counter that is greater than zero (0).
            * DO NOT send NULL message with hop counters
            * equal to zero (0).
            */
        if (Item.Item_type == MSG_ITEM || Item.Hop_Count != 0) {

            if (Param.debug&DEBUG_SERVER) /* Server tracing */
                printf("Srv %d: Sent item time %lf\n",
                    Param.ID, Item.Time_Stamp);
            (*Param.send)(Item);   /* Send the message */
        }
    }
    else {
          /* Set final stats flag */
        Srv_stats.status = STATS_FINAL;
        break;
    }

       /* Has a terminate message arrived? */
    if (c_transcount(Iam.term)) {
        Srv_stats.status = STATS_FINAL;
        break;
    }
}

   /* Calculate final statistics */
Srv_stats.ave_time_in_sys  = Stats_Mean(&sys_time);
Srv_stats.std_time_in_sys  = Stats_STD(&sys_time);
Srv_stats.max_time_in_sys  = sys_time.max_val;
Srv_stats.ave_serve_time   = Stats_Mean(&serve_stats);
Srv_stats.std_serve_time   = Stats_STD(&serve_stats);
Srv_stats.max_serve_time   = serve_stats.max_val;
Srv_stats.per_busy = (Srv_stats.sim_time) ?
    (double)(100 * sum_serve_time)/Srv_stats.sim_time : 0;
Srv_stats.q_stats =
    Param.Que.stats(Srv_stats.sim_time);
(*Param.stats)(Srv_stats);  /* Send final stats */

accept term();  /* Accept terminate message */

/* Terminate */
}
```

F.22: queue.cc

```
#include "dclr.h"
#include "queue.h"

/*****************************************************************
 *  The Queue Processes:
 *  Queuing and dequeue messages.
 *   functions:
 *     Init_Que_list  -- Initialize the Queue
 *     get_Que_item   -- Get an item from the queue using some method
 *     put_Que_item   -- Put an item into the queue
 *   processes:
 *     Queue  -- The queue process.
 *
 *****************************************************************
 * The Queue:
 *  A queue is a single linked list.  When the queue process starts
 *   up a segment of memory is malloc'ed off for the queue.  A
 *   linked list of free queue items is maintained in the Free
 *   linked list.  The actual queue is maintained by Head and Tail.
 *
 * Enqueuing:
 *  When an Item is to be Enqueued, a free node is removed from the
 *   Free list (must adjust the Free pointer, and links).  This node
 *   is attached to the tail of the queue and the item is inserted
 *   into the node.  The links must be adjusted.  If the list is
 *   empty then Head == Tail == NULL.  If there is only one (1) item
 *   in the list then Head == Tail, and Tail points to the node
 *   containing the item.
 *
 * Dequeueing:
 *
 *****************************************************************
 */
/*****************************************************************
 *  Function: Init_Que_list()
 *    qp       : pointer to Queue Structure
 *    ID       : ID of the Queue
 *    Max_Size : Size of the Queue
 *    Method   : Method of dequeueing
 *
 *  Initialize the queue. Allocate memory for the queue.  Link up
 *  the Free list.  Head & Tail = NULL.  Init. Stats. structures.
 *****************************************************************
 */
static void Init_Que_list(qp,ID,Max_Size,Method)
Queue_List  *qp;
int         ID;
int         Max_Size;
int         Method;
{
   register i;   /* Index */

   qp->ID       = ID;            /* Note Id of Queue              */
```

```
    qp->Max_Size = Max_Size;       /* Note Size of Queue             */
    qp->Method   = Method;         /* Note Queuing Method            */
    qp->Num_Elem = 0;              /* The Queue starts out as Empty */
    qp->Head     = NULL;           /* Head doesn't point anywhere   */
    qp->Tail     = NULL;           /* Tail doesn't point anywhere   */

    qp->Null_Msg_in_Q = FALSE;     /* The Null Msg Queue is Empty   */

    Stats_Init(&qp->Que_Time);     /* Init. Time in Queue Stats     */
    Stats_Init(&qp->Que_Size);     /* Init. Queue Size Stats        */

        /* Allocate Memory for the Queue. */
        /* Free points to this memory     */
    qp->Free_ptr = (Q_ITEM *)malloc(sizeof(Q_ITEM) * Max_Size);

          /* Print Warning If malloc fails */
    if (qp->Free_ptr == NULL) {
        fprintf(stderr,"Queue %d: malloc failed\n",qp->ID);
        c_exit(3);  /* Abort if malloc failed */
    }

        /* Set up a pointer to the Free list */
    qp->Free = qp->Free_ptr;

        /* Link the Free list nodes together */
    for (i=0;i < Max_Size - 1; i++) {
        qp->Free[i].Next = &(qp->Free[i+1]);
    }
        /* The Last Free list node points to NULL */
    qp->Free[i].Next = NULL;
}


/*********************************************************************
 * Function: get_Que_item()
 *    qp       : pointer to Queue Structure
 *    item     : pointer to the gotten item
 *    method   : Dequeueing Discipline
 *    sim_time : Simulation Time of calling server
 *
 * Get an item from the queue.  Return an item (in item) from the
 *   queue using the dequeueing method.  Try to return (Real)
 *   messages first. If the queue is empty and there is a Null Msg,
 *   then return the Null message.  Copy the item into the item
 *   (parameter) and attach the item_node to the head of the Free
 *   list.
 *********************************************************************
 */
static void get_Que_item(qp,item,method,sim_time)
Queue_List *qp;
ITEM        *item;
int         method;
double      sim_time;
{
   Q_ITEM *Range;   /* Range pointer     */
   Q_ITEM *ptr;     /* Temporary pointer */
```

```
int     count;     /* counter            */

   /* Are there any messages in the queue? */
if (qp->Num_Elem > 0) {

    /* Is there a NULL msg in the NULL msg queue with a timestamp
     * smaller then the current simulation time, then discard the
     * NULL messages.
     */
  if (qp->Null_Msg_in_Q && qp->Null_Queue.Time_Stamp <= sim_time)
    qp->Null_Msg_in_Q = FALSE;

    /* Determine the range of valid messages */
  for (Range = qp->Head,count=1;
    (Range->Next != NULL &&
    Range->Next->Item.Time_Stamp <= sim_time);
    Range = Range->Next,count++);

    /* Handle the case where their is either 1 item in the queue
     * or there is 1 item in the proper range.
     * if head == range == tail : only 1 item in the queue
     * if head == range != tail : only 1 item in the proper range
     */
  if (qp->Head == Range) {
    *item = qp->Head->Item;    /* Get the item                 */
    ptr   = qp->Head;          /*  ptr -> to item to free      */
    if (qp->Head == qp->Tail)  /* There is 1 item in the queue */
      qp->Head = qp->Tail = NULL;  /* Set Head & Tail to Null  */
                               /*  The Queue is now empty.     */
    else                       /* There is 1 item in the range */
      qp->Head = qp->Head->Next;   /*  Head -> to next item    */
 }
 else {
     /* Determine which queueing discipline to use */
   switch (method) {
   case LIFO:  /* Last In, First Out Method */

     *item = Range->Item;   /* Get the item */

       /* Traverse the list until ptr->Next == Range */
     for (ptr=qp->Head; ptr->Next != Range; ptr = ptr->Next);

       /* If Range == Tail : adjust tail */
     if (Range == qp->Tail) {
       qp->Tail       = ptr;    /* Tail = ptr     */
       qp->Tail->Next = NULL;   /* Fix Tail->Next */
     }
       /* If Range != Tail: link across */
     else ptr->Next = ptr->Next->Next;
     ptr = Range;  /* ptr must point to the item to be freed */
     break;

   case SIRO: /* Service In Random Order Method */
     {
       register loc,i;   /* indexes            */
       Q_ITEM   *t_ptr;  /* Temporary pointers */
```

```
      /* Get a random number loc:: 0 <= loc < count */
    loc = rand() % count;

      /* If loc == 0 : then remove the head item */
    if (loc == 0) {
      *item    = qp->Head->Item;
      t_ptr    = qp->Head;
      qp->Head = qp->Head->Next;
    }
    else {
        /* Find the loc item (ptr->Next == loc item) */
     for (ptr=qp->Head,i=1;i<loc;ptr=ptr->Next,i++);

     *item = ptr->Next->Item;    /* Get the item */
     t_ptr = ptr->Next;          /* Item to free */

       /* Are we removing the tail item? */
     if (ptr->Next == qp->Tail) {
     qp->Tail = ptr;    /* Move the tail           */
     qp->Tail = NULL;   /* Adjust the next pointer */
     }
       /* Link across */
     else ptr->Next = ptr->Next->Next;
   }
     /* t_ptr is pointing to the item to be freed  */
   ptr = t_ptr;  /* set ptr to point to item to free */
 }
 break;

case PRIO:  /* PRIority Ordering Method */
  {
    int maxprior;      /* max priority item value      */
    Q_ITEM *p_ptr;     /* pointer to max priority item */

    maxprior = qp->Head->Item.Priority;
    p_ptr    = qp->Head;

      /* Find the item in head -- Range with the max
       * Priority.  Set max_priority to priority of the
       * Head item and examine priorities through the
       * Next link. So ptr->Next == item with max
       * priority (unless max_priority is Head).
       */
    for (ptr=qp->Head; ptr != Range; ptr = ptr->Next) {
      if (ptr->Next->Item.Priority > maxprior) {
        maxprior   = ptr->Next->Item.Priority;
        p_ptr      = ptr;
      }
    }

      /* If Max priority is the Head item */
    if (p_ptr == qp->Head) {
      *item    = qp->Head->Item; /* Get the item        */
      ptr      = qp->Head;       /* ptr -> item to free */
      qp->Head = qp->Head->Next; /* Adjust Head         */
    }
```

```
          else {  /* Max Priority is ptr->Next */
            *item = p_ptr->Next->Item; /* Get the item         */
            ptr   = p_ptr->Next;       /* ptr -> item to free */

               /* Did we remove the last item? */
            if (ptr == qp->Tail) {
              qp->Tail       = p_ptr;  /* Move the Tail      */
              qp->Tail->Next = NULL;   /* Adjust Tail->Next */
            }
            else p_ptr->Next = p_ptr->Next->Next;
          }
        }
        break;

      case FIFO:  /* First In, First Out Method */
      default:    /* The default method is FIFO */
        *item    = qp->Head->Item;   /* Get the Head item      */
        ptr      = qp->Head;         /* ptr -> to item to free */
        qp->Head = qp->Head->Next;   /* Move the Head          */
        break;
      }
    }

     /* Record Queue Statistics, adjust the number in queue, &
      * Free the item pointed to by ptr (attach it to the free list)
      */
    Stats_Val(&qp->Que_Time,sim_time - item->Enter_in_Que);
    Stats_Val(&qp->Que_Size,(double)count);

    qp->Num_Elem -= 1;         /* The queue has 1 less item now   */

      /* Attach the item -> by ptr to the head of the free list */
    ptr->Next     = qp->Free;  /* ptr->Next -> to the Free list   */
    qp->Free      = ptr;       /* Free -> to ptr                  */
  }
  else {  /* Handle Null Msgs If the Queue is Empty */
    *item = qp->Null_Queue;       /* Get the null message       */
    qp->Null_Msg_in_Q = FALSE;    /* The Null queue is now empty */
  }
}

/*********************************************************************
 * Function: put_Que_item()
 *    qp       : pointer to Queue Structure
 *    item     : pointer to the item to be put in the queue
 *    sim_time : Simulation Time of calling server
 *
 * Put an item into a Queue.  Free nodes are in a list pointed to by
 *  qp->Free, qp->Head points to head of queue, qp->Tail points to
 *  tail of queue.  Items are enqueue at the tail of the queue.
 * Note: qp->Tail.Item.Enter_in_Q & qp->Num_Elem.
 * Don't forget to handle links.
 *
 * Handle Null messages separately. Do Not Queue them.  There is a
 *  Single Null message Queue for Null messages. Put incoming Null
 *  messages in this Queue,
```

```
 * Overwriting the old Null msg, Set the Null_Msg_in_Q Flag.
 *********************************************************************
 */
static void put_Que_item(qp,item,sim_time)
Queue_List *qp;
ITEM       *item;
double     sim_time;
{
    if (item->Item_type == MSG_ITEM) {  /* This is a Valid Message */

            /* Empty Queue case */
      if (qp->Head == NULL) qp->Head = qp->Free;
      else qp->Tail->Next = qp->Free;      /* Otherwise case       */

      qp->Tail       = qp->Free;           /* Get a free Item node */
      qp->Free       = qp->Free->Next;     /* Fix up Free list     */
      qp->Tail->Next = NULL;
      qp->Tail->Item = *item;              /* Put Item in Queue    */
      qp->Tail->Item.Enter_in_Que = sim_time; /* Record Entry Time */
      qp->Num_Elem += 1;                      /* Note: Queue Count */
    }
    else {   /* This is a Null Message */
      qp->Null_Queue    = *item; /* Put Null Msg in Null Queue */
      qp->Null_Msg_in_Q = TRUE;  /* There is a Que'd Null Msg  */
    }
}


/*********************************************************************
 * The Queue Process.
 *   The queue process accepts incoming messages, and buffers them
 *   until the associated server is ready to process them.  When the
 *   queue is full, the queue will refuse to accept incoming msgs.
 *********************************************************************
 */
process body Queue()
{
    process     Queue Iam;            /* Queue's process id        */
    struct      q_param Param;        /* Queue's parameter table   */
    struct      q_stats_rec q_stats;  /* Queue's statistics table  */
    Q_ITEM      *Range;               /* pointer into the queue    */
    Queue_List  Q;                    /* the queue buffer          */
    double      Time = 0.0;           /* simulation time           */
    int         Count;                /* Count                     */
    int         Done = FALSE;         /* Flag                      */
    ITEM        Res_item;             /* Resolver NULL message     */
    ITEM        item;                 /* Normal message            */

      /* Get Queue's process id */
    Iam = (process Queue)c_mypid();

      /* Initialize the Resolver NULL message */
    Res_item.Time_Stamp = 0.0;
    Res_item.Item_type  = MSG_NULL;

      /* Get Queue's parameter table */
```

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

  /* create a queue buffer */
Init_Que_list(&Q,Param.ID, Param.Q_size, Param.Q_method);

for (;;) {
    if (Param.debug&DEBUG_QUEUE) { /* Queue tracing */
       printf("Queue %d: #In %d c_tc.Send %d c_tc.Res_send %d\n",
            Param.ID,Param.Num_Fan_In, c_transcount(Iam.send),
            c_transcount(Iam.Res_send));
       printf("Done %d  Q.Num_Elem %d  Param.Q_size  %d\n",
            Done,Q.Num_Elem, Param.Q_size);
    }

    bump_seq();  /* For Scott's Deadlock Detectors */
    select {
           /* If there is a message in the Queue, and the
            * server request a message, then give it to the
            * server
            */
        (!Done && (Q.Num_Elem > 0 || Q.Null_Msg_in_Q)):
           accept get_item(Item,sim_time)
        {
            Time = sim_time;
            get_Que_item(&Q,Item,Param.Q_method,Time);
        };

        /* During termination, return a "Error" to the server */
    or (Done):
        accept get_item(Item,sim_time)
        {   Item->Item_type = MSG_ERR; };

       /* If the queue is not full */
    or (!Done && Q.Num_Elem < Param.Q_size):

          /* Accept a normal message, if there are Num_Fan_In
           * outstanding messages and there are no Resolver
           * NULL messages.
           */
        accept send (Item)
        suchthat (c_transcount(Iam.send) == Param.Num_Fan_In &&
                  c_transcount(Iam.Res_send) == 0)
        by (Item.Time_Stamp)
        {   item = Item; };
            put_Que_item(&Q,&item,Time);

       /* During terminations, discard all incoming messages */
    or (Done):
        accept send (Item) {};

      /* Accept a Resolver NULL message if there are Num_Fan_In
       * outstanding normal and Resolver NULL messages.
       */
    or  (!Done):
        accept Res_send (Item)
        suchthat (c_transcount(Iam.send) +
```

```
            c_transcount(Iam.Res_send) >=
            Param.Num_Fan_In)
    by (Item.Time_Stamp)
    { Res_item = Item; };

    /* Accept all other outstanding Resolver NULL messages */
    while (c_transcount(Iam.Res_send) > 0)
        accept Res_send(Item) {};

       /* If the Queue is not Full */
    if (Q.Num_Elem < Param.Q_size ) {
          /* Determine if there are any normal messages
           * with a timestamp with a smaller value
           */
        select {
            accept send (Item)
            suchthat (Item.Time_Stamp < Res_item.Time_Stamp)
            by (Item.Time_Stamp)
            { item = Item ; };
        or
            item = Res_item;
        }
              /* put the message in the queue */
            put_Que_item(&Q,&item,Time);
    }

or
      /* The server will request the stats from the queue */
    accept stats(sim_time) {

           /* Calculate and return stats */
        for (Range = Q.Head,Count=0; Range != NULL &&
            Range->Item.Time_Stamp <= sim_time;
            Range = Range->Next,Count++);

        q_stats.num_through_q = Q.Que_Time.num_val;
        q_stats.ave_time_in_q = Stats_Mean(&Q.Que_Time);
        q_stats.std_time_in_q = Stats_STD(&Q.Que_Time);
        q_stats.max_time_in_q = Q.Que_Time.max_val;
        q_stats.ave_in_q = Stats_Mean(&Q.Que_Size);
        q_stats.std_in_q = Stats_STD(&Q.Que_Size);
        q_stats.max_in_q = Q.Que_Size.max_val;
        q_stats.per_full = (double)(100*Count)/Param.Q_size;
        q_stats.num_in_q = Count;
        treturn (q_stats);
    };

or
      /* accept the terminate message */
    accept term() {};
    Done = TRUE;
    free((char *)Q.Free_ptr);

or (Done):
    terminate;  /* Terminate */
}
```

```
        c_sch(); /* swh : schedule another process */
    }
}
```

F.23: sink.cc

```
#include "dclr.h"

/*******************************************************************
 * sink.cc -- by Edward Vopata
 *******************************************************************
 * Accept incoming messages and discard them.  Keep track of the
 * number of discarded messages and report this statistics to the
 * collector.  May have to timeout and send stats periodically, in
 * case the sink received the final generated message.
 *******************************************************************
 */

void do_sink();  /* the do_sink() function */

/*******************************************************************
 * Function:
 *   do_sink()
 * Parameter:
 *   item  - the item to be discarded
 *   Param - Sink's Parameter table
 *   Stats - Sink's Stats table
 *   i_num - Inteval number
 * Summary:
 *   discard the item and record and report stats as needed.
 *******************************************************************
 */
void do_sink(item,Param,Stats,i_num)
ITEM   *item;
struct sink_param     *Param;
struct sink_stats_rec *Stats;
long   *i_num;
{
    if (Param->debug&DEBUG_SINK)  /* Sink tracing */
        printf("Sink %d: Item arrived Time %lf\n",
            Param->ID, item->Time_Stamp);

      /* Is it time to send stats? Is I_num different then the
       * current interval
       */
    if (*i_num != (long)(item->Time_Stamp / Param->stats_interval))
    {
          /* If so, update i_num and send stats to the collector */
        *i_num  = (long)(item->Time_Stamp / Param->stats_interval);
        (*Param->stats)(*Stats);

        if (Param->debug&DEBUG_SINK) /* Sink tracing */
            printf("Sink %d: Sent Stats\n", Param->ID);
    }
      /* Keep track of the number of Normal messages discarded */
    if (item->Item_type == MSG_ITEM) Stats->num_sunk += 1;

      /* Keep track of simulation time */
    if (item->Item_type == MSG_ITEM && item->Time_Stamp != MAX_TIME)
```

```
        Stats->sim_time = MAX(item->Time_Stamp,Stats->sim_time);
}

/*******************************************************************
 * The Sink Process:
 *   Accept incoming messages and call do_sink().
 *   Handle both normal and Resolver NULL messages.
 *   The Sink represent a point in the physical system where jobs
 *     leave the system.
 *******************************************************************
 */

process body Sink ()
{
    struct  sink_stats_rec Sink_stats; /* Sink Statistics          */
    struct  sink_param     Param;      /* Sink Parameter List      */
    process Sink Iam;                  /* Id for c_transcount      */
    long    i_num = 0;                 /* Current Stats interval   */
    int     Done  = FALSE;             /* Termination Flag         */
    ITEM    Res_item;                  /* Resolver NULL message    */
    ITEM    item;                      /* Normal message           */

    Iam = (process Sink)c_mypid();  /* Get Sink's process id */

       /* Initialize Res_item */
    Res_item.Time_Stamp = 0.0;
    Res_item.Item_type  = MSG_NULL;

      /* accept Paramter table */
    accept setup(Snk) {Param = Snk; };

      /* Initialize Stats report */
    Sink_stats.ID = Param.ID;
    Sink_stats.sim_time = 0;
    Sink_stats.num_sunk = 0;
    Sink_stats.status   = STATS_NORMAL;

    for (;;) {
        bump_seq();   /* For Scott's Deadlock Detection */
        select {

        (!Done):
               /* Accept a normal message if there are Num_Fan_In
                * outstanding messages and NO Resolver NULL msgs
                * by the smallest timestamp.
                */
            accept send(Item)
            suchthat(c_transcount(Iam.send) == Param.Num_Fan_In &&
                     c_transcount(Iam.Res_send) == 0)
            by (Item.Time_Stamp)
            { item = Item; } ;

             if (Param.debug&DEBUG_SINK) /* Sink tracing */
               printf("Sink %d: !Done: Real Item arrived Time %lf\n",
                 Param.ID, item.Time_Stamp);
```

```
        /* Deal with the message */
    do_sink(&item,&Param,&Sink_stats,&i_num);

or (Done):
       /* During termination discard all incoming messages */
    accept send(Item) {};

or (!Done):
       /* Accept Resolver NULL messages if there are
        * NUM_FAN_IN normal and Resolver NULL msgs, by
        * smallest timestamp
        */
    accept Res_send(Item)
    suchthat(c_transcount(Iam.send) +
             c_transcount(Iam.Res_send) >=
             Param.Num_Fan_In)
    by (Item.Time_Stamp)
    {   Res_item = Item;};

       /* Accept all other Resolver NULL msgs */
    while (c_transcount(Iam.Res_send) > 0)
        accept Res_send(Item) {};

    if (Param.debug&DEBUG_SINK) /* Sink tracing */
       printf("Sink %d: !Done: Item arrived Time %lf\n",
          Param.ID, Res_item.Time_Stamp);

       /* Determine if there is a normal message with a
        * smaller timestamp.  Deal with the messages by
        * calling do_sink().
        */
    select {
       accept send (Item)
       suchthat (Item.Time_Stamp < Res_item.Time_Stamp)
       by (Item.Time_Stamp)
       { item = Item; };
       if (Param.debug&DEBUG_SINK) /* Sink tracing */
         printf("Sink %d:Real send: Item arrived Time %lf\n",
            Param.ID, item.Time_Stamp);
    or
       item = Res_item;
       if (Param.debug&DEBUG_SINK) /* Sink tracing */
       {
          printf("Sink %d:Dealing with Res:",Param.ID);
          printf("Item arrived Time %lf\n",item.Time_Stamp);
       }
    }
       /* Deal with the message */
    do_sink(&item,&Param,&Sink_stats,&i_num);

or
       /* Accept terminate message */
    accept term() {};
    if (Param.debug&DEBUG_SINK)  /* Sink tracing */
        printf("Sink %d: Terminate Mode\n", Param.ID);
```

```
            /* Send final Stats */
          Done = TRUE;
          Sink_stats.status = STATS_FINAL;
          (*Param.stats)(Sink_stats);
          if (Param.debug&DEBUG_SINK)
              printf("Sink %d: Terminate Mode\n", Param.ID);

      or (!Done):
             /* timeout & send stats */
          delay 2;  /* Delay 2 seconds */
          (*Param.stats)(Sink_stats);

      or (Done):
          terminate;  /* Terminate */
      }
  }
}
```

F.24: branch.cc

```
#include "dclr.h"

/*****************************************************************
 * branch.cc -- by Edward Vopata
 *
 * The branch logical process.
 *    accept incoming messages, select an outgoing line and send the
 *    message.  Deal with Resolver NULL messages.  Return the current
 *    simulation time of the branch.
 *****************************************************************
 */

static void do_branch();

/*****************************************************************
 * Function: do_branch
 * Parameter:
 *      item  -- The incoming messages
 *      Param -- The Parameter of the Branch
 *      Time  -- The Simulation Clock
 *      i_num -- Interval number
 * Purpose:
 *  This function takes a message (item) and determines which
 *  outgoing path to send the message on and sends it.  This function
 *  also propagates a null message on all other outgoing paths when
 *  there is an interval change, in order to propagate the simulation
 *  time and aid in statistics collection.
 *****************************************************************
 */

static void do_branch(item,Param,Time,i_num)
ITEM     *item;               /* Incoming Messages */
struct   brn_param *Param;    /* Branch Parameters */
double   *Time;               /* Simulation Clock  */
long     *i_num;              /* Interval Number   */
{
    register i,j;                /* Index Variables          */
    double   prb;                /* Line selection Variable  */
    double   rnd;                /* Line selection Variable  */

        /* Update the Branch's Simulation Clock */
    *Time = item->Time_Stamp;

        /* Print Time Stamp of Message if Debug Mode On */
    if (Param->debug&DEBUG_BRANCH)  /* Debugging */
        printf("Branch %d: Got an Item Time = $ld\n",
            Param->ID,*Time);

        /* Determine which outgoing line to send the message on.
         * 1. Select a random number
         * 2. Divide the path probabilities into ranges
         * 3. Determine in which range the random number falls
         * 4. Send the message on that path.
```

```
     */
rnd = drand01();    /* Get a uniform random number */
prb = 0.0;          /* Initialize the lower range  */

     /* Step through the list of possible paths */
for (i=0; i < Param->Num_Fan_Out; i++) {

        /* Is the random number in the range of
         * prb <= rnd < Path.prob + prb?
         */
    if (prb <= rnd && rnd < Param->Fan_Out[i].prob + prb) {

            /* If So, Send the message */
        (*Param->Fan_Out[i].send)(*item);

            /* If Debug Mode, Print Timestamp */
        if (Param->debug&DEBUG_BRANCH)
            printf("Branch %d: Send Item %d\n",
                Param->ID,Param->Fan_Out[i].ID);

            /* We Found the Correct Path,
             * so break out of the loop
             */
        break;
    }
        /* Otherwise adjust the lower range to
         * the lower range plus the Path.prob
         */
    prb += Param->Fan_Out[i].prob;
}
/* Example of Range calculation:
 * Three out going paths, with probabilities 0.2, 0.3, and 0.5.
 * rnd is the selected random number.
 * Range check  : prb <= rnd < Path.prob + prb
 * Range check 1: prb = 0.0, Path.prob = 0.2
 *       Range  : 0.0 <= rnd < 0.2 + 0.0
 *  If rnd is in range 0.0 to 0.2 then path 1 is selected
 *       update : prb = prb + Path.prob = 0.0 + 0.2 = 0.2
 * Range check 2: prb = 0.2, Path.prob = 0.3
 *       Range  : 0.2 <= rnd < 0.3 + 0.2
 *  If rnd is in range 0.2 to 0.5 then path 2 is selected
 *       update : prb = prb + Path.prob = 0.2 + 0.3 = 0.5
 * Range check 2: prb = 0.5, Path.prob = 0.5
 *       Range  : 0.5 <= rnd < 0.5 + 0.5
 *  If rnd is in range 0.5 to 1.0 then path 3 is selected
 *
 ************************************************************
 * Note the PATH probabilities MUST sum to 1.0.
 ************************************************************
 */

    /* Send nulls on other lines at interval transisition
     * This is used to propagate statistics collection and
     * to do some minor deadlock avoidance.
     */
```

```
        /* Is this a new interval? */
    if (*i_num != (long)(item->Time_Stamp / Param->stats_interval))
    {
            /* If Debug Mode, print timestamp */
        if (Param->debug&DEBUG_BRANCH)
            printf("Branch %d: Sending NUll %lf\n",
                Param->ID,item->Time_Stamp);

            /* Calculate the new interval number */
        *i_num = (long)(item->Time_Stamp / Param->stats_interval);

            /* Build a Null Message */
        item->Arrive_Time = item->Time_Stamp; /* Same Timestamp    */
        item->Item_type   = MSG_NULL;         /* a Null MSG        */
        item->Hop_Count   = Param->Max_Hop_Count; /* Hop Counter   */
        item->Id_of_Src   = Param->ID;        /* Id of the Branch  */
        item->data[0]     = '\0';             /* No extra data     */

            /* Send the Null MSG on all paths except the one
             * that the original message was sent on.
             */
        for (j=0;j < Param->Num_Fan_Out; j++)
          if (j != i) (*Param->Fan_Out[i].send)(*item);
    }
}

process body Branch ()
{
    process  Branch Iam;          /* Process Id of the Branch  */
    struct   brn_param Param;     /* Branch Parameters         */
    double   Time     = 0.0;      /* Simulation Clock          */
    long     i_num    = 0;        /* Interval number           */
    int      Done     = FALSE;    /* Flag                      */
    ITEM     Res_item;            /* Resolver Message          */
    ITEM     item;                /* Incoming/Outgoing Message */

    srand(getpid() * getppid());        /* Seed the random number  */
    Iam  = (process Branch)c_mypid();   /* Get Branch's Process Id */

        /* Create a phony null message */
    Res_item.Time_Stamp = 0.0;
    Res_item.Item_type  = MSG_NULL;

    accept setup(Brn) { Param = Brn; }; /* Get Branch's Parameters */

        /* Loop Forever */
    for (;;) {

        bump_seq();   /* For Scott's deadlock detection */
        select {
           /* accept a normal messages, if there are Num_Fan_In
            * messages and NO resolver NULL messages, by smallest
            * timestamp, and deal with the messages (do_branch()).
            */
        (!Done):
            accept send(Item)
```

```
    suchthat (c_transcount(Iam.send) == Param.Num_Fan_In &&
              c_transcount(Iam.Res_send) == 0)
    by (Item.Time_Stamp)
    {   item = Item; };
        do_branch(&item,&Param,&Time,&i_num);

   /* At termination, receive all incoming messages and
    * discard them.
    */
or (Done):
    accept send(Item) {};

   /* Receive a resolver NULL messages, if there are
    * Num_Fan_In normal and Resolver messages, by smallest
    * timestamp
    */
or (!Done):
    accept Res_send(Item)
    suchthat (c_transcount(Iam.send) +
              c_transcount(Iam.Res_send) >=
              Param.Num_Fan_In)
    by (Item.Time_Stamp)
    {   Res_item = Item; };

      /* Discard the rest of the Resolver NULL messages */
    while (c_transcount(Iam.Res_send) > 0)
        accept Res_send(item) {};

      /* Determine if there is a normal message with a
       *  smaller timestamp, if so, get it.  then deal
       *  with the message, (do_branch()).
       */
    select {
        accept send (Item)
        suchthat (Item.Time_Stamp < Res_item.Time_Stamp)
        by (Item.Time_Stamp)
        { item =Item; };
    or
        { item = Res_item; }
    }
    do_branch(&item,&Param,&Time,&i_num);

or
      /* Return the current Time */
    accept get_time() {treturn Time; };
    if (Param.debug&DEBUG_BRANCH)  /* Debugging */
          printf("Branch %d: Time %lf\n", Param.ID, Time);

or
      /* accept the terminate messages */
    accept term() {}; Done = TRUE;

or (Done):
    terminate;  /* Terminate alternative */
}
c_sch(); /* swh : schedule another process */
```

```
        }
    }
```

F.25: col.cc

```
#include "dclr.h"


/**********************************************************************
 * col.cc -- by Edward Vopata
 **********************************************************************
 * The Collector Process.
 *   Collects stats reports from the logical processes and compiles
 *   a Collective Stats Report.  The Collective Stats Report is
 *   sent to the Graphics Front-end.
 **********************************************************************
 */
   /* Status of the Collector */
#define NOT_DONE  0
#define TERM_TIME 1
#define ALL_SUNK  2

void   M_Error();
struct col_rec *get_rec();
struct col_rec *get_ptr();
void   free_rec();
void   adjust_rec();
void   Send_Stats();
void   Print_Stats();
int    Wait_resp();

#define E_MALLOC 1 /* Error return code: Malloc Failed */

  /* Malloc Failed Error Function */
static void M_Error(sock,err_msg)
int  sock;
char *err_msg;
{
       /* If malloc fails, the print an error msg, and
        * send the graphics front--end a "(abort)" msg.
        */
    fprintf(stderr,"%s\n",err_msg);
    putline(sock,"(abort)",FALSE);
    c_exit(E_MALLOC);
}

   /* Create a new stats report record */
static struct col_rec *get_rec(i_num,sock)
long i_num;
int  sock;
{
    struct col_rec *ptr;

      /* malloc off a new stats report record */
    ptr = (struct col_rec *) malloc (sizeof(struct col_rec));
    if (ptr == NULL) M_Error(sock, "Get_rec: Malloc Failed");
       /* Initialize the record. */
    ptr->i_num   = i_num;
```

```
     set_clear(ptr->modified);
     bzero(ptr->Stats, sizeof(ptr->Stats));
     ptr->Next    = NULL;
     return ptr;
}
   /* Get a pointer to the requested stats record */
static struct col_rec *get_ptr(head,tail,i_num,sock)
struct col_rec *head;
struct col_rec **tail;
long   i_num;
int    sock;
{
     struct col_rec *ptr;
     register i;

          /* Are the pointers already pointer to the
           * desired record? If so, return the pointer
           */
     ptr = *tail;
     if (head->i_num == i_num) return (head);
     if (ptr->i_num  == i_num) return (ptr);

          /* Otherwise search the list and find the
           * correct record
           */
     if (ptr->i_num < i_num) {
          for (i=ptr->i_num + 1; i <= i_num; i++) {
               *tail     = get_rec(i_num,sock);
               ptr->Next = *tail;
               ptr       = *tail;
          }
          return (*tail);
     }

     ptr = head;
     while ((ptr = ptr->Next) != NULL)
          if (ptr->i_num == i_num) return (ptr);

     return (NULL);
}


   /* Free up a used stats record */
static void free_rec(ptr)
struct col_rec *ptr;
{
     register i;

     ptr->Next = NULL;
     for (i=0; i < MAXPROC; i++) {
          if (ptr->Stats[i].free_stats != NULL) {
               free ((char *) ptr->Stats[i].free_stats);
          ptr->Stats[i].free_stats = NULL;
          }
     }
     free((char *)ptr);
```

```
}

  /* clean of the stats list after sending stats to the
   * graphics front-end.  Free up the used record
   */
static void adjust_rec(head,tail,Send,Final,sock)
struct col_rec **head, **tail;
SET    Send, Final;
int    sock;
{
    struct col_rec *ptr;
    int i_num;

    ptr = *head;
    *head = ptr->Next;
    i_num = ptr->i_num;      /* unlink the record       */
    free_rec (ptr);          /* and free it             */
    set_clear(Send);         /* Clear the Send Set      */
    set_union(Send,Final);   /* Note any Finalized LP's */
    if (*head == NULL) {
        *head = *tail = get_rec(i_num + 1,sock);
    }
    else {
        ptr = *head;
        while (ptr != NULL) {
            set_union(Send,ptr->modified);
            ptr = ptr->Next;
        }
    }
}


/*******************************************************/

/* Send stats to the graphics front end.  Send the stats
 * record that is at the head of the stats list
 */
static void Send_Stats(Param,head,status)
struct col_param *Param;
struct col_rec *head;
int    status;
{
    char line[MAXLINE];
    register i, sock;

    sock = Param->PS.out_sock;

       /* Do we have to send status this time? */
    switch (status) {
        case TERM_TIME:
        case ALL_SUNK:
            putline(sock,"(end)",Param->PS.debug&DEBUG_SOCK);
            break;

        case NOT_DONE:
        default:
```

```
        break;
}

   /* Send simulation time */
if (status == NOT_DONE || Param->PS.sim_term_time == 0.0)
    sprintf(line,"(%.4lf)",
        (head->i_num * Param->PS.stats_interval) +
        Param->PS.stats_interval);
else
    sprintf(line,"(%.4lf)",Param->PS.sim_term_time);

putline(sock,line,Param->PS.debug&DEBUG_SOCK);

   /* For each logical process in the process table */
for (i=0; i < Param->PS.num_nodes; i++) {
       /* If the stats record was modified by the
        * LP, then send its stats to the graphics front-end */
    if (set_in(head->modified,i)) {
        switch (Param->PS.type[i]) {
        case SOURCE:  /* The Source LP */
            sprintf(line,"(%d %ld)",i,
                head->Stats[i].Src_stats->num_left);
            break;

        case QUE_SRV:  /* The Queue & Server LP */
            sprintf(line,"(%d %.4lf %.4lf %ld %ld)",i,
               head->Stats[i].Q_Srv_stats->per_busy,
               head->Stats[i].Q_Srv_stats->q_stats.per_full,
               head->Stats[i].Q_Srv_stats->q_stats.num_in_q,
               head->Stats[i].Q_Srv_stats->num_served);
            break;

        case SINK:  /* The Sink LP */
            sprintf(line,"(%d %ld)",i,
                head->Stats[i].Sink_stats->num_sunk);
            break;

        default:
            fprintf(stderr,"Send_Stats: Invalid type (%d)\n",
                Param->PS.type[i]);
            break;
        }
           /* send the report */
        putline(sock,line,Param->PS.debug&DEBUG_SOCK);
    }
}
    /* send the "($$)" to denote the end of the current
     * report
     */
putline(sock,"($$)",Param->PS.debug&DEBUG_SOCK);

   /* If Switch #14 is sent then record all the statistics
    * in the Stats_Report File.
    */
if (Param->PS.debug&STATS_REPORT)
    Print_Stats(Param,head,status);
```

```
}

/* Wait for a response from the graphics front end */
static int Wait_resp(Param)
struct col_param *Param;
{
    register Done;
    int      type, ID;
    char     line[MAXLINE];  /* Incoming line */
    char     *ptr;

      /* Get a line from the graphics front-end (blocking) */
    getline(Param->PS.in_sock,line,Param->PS.debug&DEBUG_SOCK);
    ptr = index(line,'(') + 1;
    ptr = index(ptr,'(') + 1;
    sscanf(ptr,"%d %d",&type,&ID);
    if (Param->PS.debug&DEBUG_SOCK)
        printf("Wait_resp: %s\n",line);

      /* What type of control message was it? */
    switch(type) {

        case SIG_CONT:    /* It was a continue message */
            Done = FALSE;
            break;

        case SIG_TERM:  /* It was a terminate message */
            Param->PS.Term_pid.term();
            Done = TRUE;
            break;

        default:
            fprintf(stderr,"Wait_resp: Invalid Type (%d)\n",type);
            Done = FALSE;
            break;
    }
    return Done;
}

/* the Collector Process
 * receives stats reports from each logical process and keeps a
 * list of the report,  when a complete report is obtained the
 * report is sent to the graphics front-end.
 */

process body Collector ()
{
    struct  col_param *Param;         /* Col parameter table        */
    struct  col_rec *head,*tail,*ptr; /* pointer to the stats list */
    SET     Send;                     /* used to determine is a    */
                                      /* stats report is complete  */
    SET     Final;                    /* used to indicate LP's     */
                                      /* have completed.           */
    int     Done = NOT_DONE;
    double  Time;
    long    Total_Sunk;
```

```
long    Sunk[MAXPROC];
int     Sink_Flag = FALSE;
long    i_num;
register ID, status, i;

     /* create a parameter table and accept the table */
Param = (struct col_param *) malloc (sizeof(struct col_param));
accept setup (col) {(*Param) = col; };

if (Param->PS.debug&DEBUG_COLLECT) /* Collector tracing */
    printf("Col: Collection Starting\n");

  /* Initialize the stats list */
bzero(Sunk,sizeof(Sunk));
head = tail = get_rec(0,Param->PS.out_sock);
set_clear (Send);
set_clear (Final);
for (i=0; i < Param->PS.num_nodes; i++)
    if (Param->PS.type[i] == BRANCH) set_add(Final,i);
set_union(Send,Final);

for (;;) {

    if (Param->PS.debug&DEBUG_COLLECT)
        printf("Col: Collection Looping\n");

      /* accept reports from each LP.
       * record the stats in the stats list.
       * don't forget to modify SEND and FINAL
       * as needeed.
       */
    select {
       /* Collect Source Stats */
        accept Src_stats (src_stats)
        {
            if (Param->PS.debug&DEBUG_COLLECT)
                printf("Col:Src %d  status %d time %.4lf\n",
                    src_stats.ID, src_stats.status,
                    src_stats.sim_time);
            ID     = src_stats.ID;
            i_num  = (long)(src_stats.sim_time /
                Param->PS.stats_interval);
            status = src_stats.status;
            Time = src_stats.sim_time;
            ptr = get_ptr(head,&tail,i_num, Param->PS.out_sock);
            if (status == STATS_FINAL || status == STATS_TERM)
                set_add(Final, ID);
            if (ptr == NULL) treturn;

            if (ptr->Stats[ID].Src_stats == NULL) {
                ptr->Stats[ID].Src_stats =
                    (struct src_stats_rec *)
                    malloc (sizeof (struct src_stats_rec));
                if (ptr->Stats[ID].Src_stats == NULL) {
                    M_Error(Param->PS.out_sock,
                        "Col:Src_Stats: Malloc Failed");
```

```
                }
            }
            *(ptr->Stats[ID].Src_stats) = src_stats;
        };
        if (ptr != NULL) {
            set_add(ptr->modified, ID);
            set_add(Send, ID);
        }
            if (Param->PS.debug&DEBUG_COLLECT)
                printf("Col: Source Stats\n");

      /* Collect Queue and Server Stats */
    or accept Que_Srv_stats (q_srv_stats)
        {
            if (Param->PS.debug&DEBUG_COLLECT)
                printf("Col:Q/Srv %d  status %d time %.41f\n",
                    q_srv_stats.ID, q_srv_stats.status,
                    q_srv_stats.sim_time);
            ID     = q_srv_stats.ID;
            i_num  = (long)(q_srv_stats.sim_time /
                Param->PS.stats_interval);
            status = q_srv_stats.status;
            Time = q_srv_stats.sim_time;
            ptr = get_ptr(head, &tail, i_num,Param->PS.out_sock);
            if (status == STATS_FINAL || status == STATS_TERM)
                set_add(Final, ID);
            if (ptr == NULL) treturn;

            if (ptr->Stats[ID].Q_Srv_stats == NULL) {
                ptr->Stats[ID].Q_Srv_stats =
                    (struct q_srv_stats_rec *)
                    malloc (sizeof (struct q_srv_stats_rec));
                if (ptr->Stats[ID].Q_Srv_stats == NULL) {
                    M_Error(Param->PS.out_sock,
                        "Col:Q_Srv_Stats: Malloc Failed");
                }
            }
            *(ptr->Stats[ID].Q_Srv_stats) = q_srv_stats;
        };
        if (ptr != NULL) {
            set_add(ptr->modified, ID);
            set_add(Send, ID);
        }
        if (Param->PS.debug&DEBUG_COLLECT)
            printf("Col: Queue/Server Stats\n");

      /* Collect Sinks Stats */
    or accept Sink_stats (sink_stats)
        {
            if (Param->PS.debug&DEBUG_COLLECT)
                printf("Col:Sink %d  status %d time %.41f\n",
                    sink_stats.ID, sink_stats.status,
                    sink_stats.sim_time);
            ID     = sink_stats.ID;
            i_num  = (long)(sink_stats.sim_time /
                Param->PS.stats_interval);
```

```
        status = sink_stats.status;
        Time = sink_stats.sim_time;
        ptr = get_ptr(head, &tail, i_num,Param->PS.out_sock);
        if (status == STATS_FINAL || status == STATS_TERM)
            set_add(Final, ID);

        if (ptr != NULL) {
            if (ptr->Stats[ID].Sink_stats == NULL) {
                ptr->Stats[ID].Sink_stats =
                    (struct sink_stats_rec *)
                    malloc (sizeof (struct sink_stats_rec));
                if (ptr->Stats[ID].Sink_stats == NULL) {
                    M_Error(Param->PS.out_sock,
                        "Col:Sink_Stats: Malloc Failed");
                }
            }
            *(ptr->Stats[ID].Sink_stats) = sink_stats;
        }
    };

       /* Keep track of the number of message sunk
        * by all the sinks.  If all the messaes have
        * been sunk
        */
    if (ptr != NULL) {
        set_add(ptr->modified, ID);
        set_add(Send, ID);
        Sunk[ID]  = ptr->Stats[ID].Sink_stats->num_sunk;
        Sink_Flag = TRUE;
    }

or accept term() {};  /* accept the terminate messages */

or (Done): delay 1;
}


if (!Done) {
       /* Send start termination message to terminator */
    if (status == STATS_TERM) {
        Param->PS.Term_pid.term();
        Done = TERM_TIME;
        continue;
    }


       /* Determine if all the messages have been sunk */
    if (!Param->PS.Infinite_Src && Sink_Flag) {
        Total_Sunk = 0;
        for (i=0; i < Param->PS.num_nodes; i++)
            Total_Sunk += Sunk[i];
        Sink_Flag = FALSE;

        if (Param->PS.debug&DEBUG_COLLECT)
            printf("Col: T_Sunk %ld  T_Gen %ld\n",
                Total_Sunk,Param->PS.Total_Gen);
```

```
                /* If all the messages are sunk, then
                 * send start termination message
                 * to terminator
                 */
            if (Total_Sunk == Param->PS.Total_Gen) (
                Param->PS.Term_pid.term();
                Done = ALL_SUNK;
                continue;
            }
        }

           /* If a complete stats report exist,
            * Send the stats to the graphics front-end
            * and await its response.  If Wait_Resp
            * return TRUE, then the Graphics front-end
            * has sent a terminate control messages
            */
        if (set_full(Send,Param->PS.num_nodes)) {
           if (Param->PS.debug&DEBUG_COLLECT)
               printf("Col: Sending Stats : (%d)\n",Done);

           Send_Stats(Param,head,NOT_DONE);
           adjust_rec(&head,&tail,Send,Final,Param->PS.out_sock);

           if (Wait_resp(Param)) break;

           if (Param->PS.debug&DEBUG_COLLECT)
               printf("Col: Sending Stats : (%d)\n",Done);
        }
    }
    else (
            /* send The FINAL stats report */
        if (set_full(Final,Param->PS.num_nodes)) (
            if (head != tail) (
                Send_Stats(Param,head,NOT_DONE);
                adjust_rec(&head,&tail,Send,Final,
                    Param->PS.out_sock);
                if (Wait_resp(Param)) break;
                if (Param->PS.debug&DEBUG_COLLECT)
                    printf("Col: Sending Stats : (%d)\n",Done);
            }
            else (
                Send_Stats(Param,head,Done);
                if (Param->PS.debug&DEBUG_COLLECT)
                    printf("Col: Sending Final Stats\n");
                break;
            }
        }
    }
}
  /* Clean up the used stats record */
while (head != NULL) (
    ptr = head;
    head = head->Next;
    free_rec(ptr);
}
```

```
    /* During Termination */
     /* Clean up */
   free((char *) Param);

       /* Discard any outstanding stats reports */
   for (;;) {
       select {
          accept Src_stats (src_stats) {};
       or accept Que_Srv_stats (q_srv_stats) {};
       or accept Sink_stats (sink_stats) {};
       or accept term() {};
       or terminate;
       }
   }
}
```

F.26: term.cc

```
#include "dclr.h"
#include "graph.h"
#include "Pid.h"
#include "defs.h"
#include "spec.h"

/*********************************************************************
 * term.cc -- by Edward Vopata
 *********************************************************************
 *  The Terminator process:
 *   1. Waits for the collector to report termination.
 *   2. Send terminate message to every Logical process
 *   3. Send terminate message to every Resolver
 *   4. Send terminate message back to the collector
 *   5. Terminate.
 *********************************************************************
 */
process body Terminator ()
{
    struct   term_param   Param;   /* Terminator Parameter Table   */
    process  resolver     Res[20]; /* List of Resolver Processes   */
    int      num_Res;              /* Number of Resolver Processes */
    register i;                    /* Index                        */

       /* Receive the Parameter Table and the List of Resolvers */
    accept setup (term) { Param = term; };
    accept setup1 (num,res) {
        num_Res = num;
        for (i=0; i < num_Res; i++)
            Res[i] = (process resolver)res[i];
    };

    for (;;) {
        select {
                /* Wait for the collector to report termination */
            accept term() {};
                if (Param.PS.debug&DEBUG_TERM)  /* Tracing */
                    printf("Term: Terminate Recv'd\n");

                    /* For each logical process, send a
                     * terminate message
                     */
                for (i=0; i<Param.PS.num_nodes; i++) {
                    switch (Param.PS.type[i]) {
                    case SOURCE:  /* To Source LP */
                        Param.PS.ps_pid[i].Src_pid.term();
                        if (Param.PS.debug&DEBUG_TERM)
                            printf("Term: Source %d Term\n",i);
                        break;
                    case BRANCH:  /* To Branch LP */
                        Param.PS.ps_pid[i].Branch_pid.term();
                        if (Param.PS.debug&DEBUG_TERM)
                            printf("Term: Branch %d Term\n",i);
```

```
                    break;
                case SINK:  /* To Sink LP */
                    Param.PS.ps_pid[i].Sink_pid.term();
                    if (Param.PS.debug&DEBUG_TERM)
                        printf("Term: Sink %d Term\n",i);
                    break;

                case QUE_SRV:  /* to Queue and Server LP */
                    Param.PS.Que_pid[i].term();
                    Param.PS.ps_pid[i].Srv_pid.term();
                    if (Param.PS.debug&DEBUG_TERM)
                        printf("Term: Queue/Server %d Term\n",i);
                    break;
                default:
                    fprintf(stderr,"Term: Invalid Type (%d)\n",
                        Param.PS.type[i]);
                }
            }

                /* Send a terminate message to each Resolver */
            for (i=0; i < num_Res; i++)
                Res[i].term();

                /* Tell Collector That Termination is Complete */
            Param.PS.Col_pid.term();

        or terminate;  /* Terminate */
        }
    }
}
```

F.27: resolver.cc

```
#include "dclr.h"
#include "graph.h"
#include "Pid.h"      /* Scott's Pid includes */
#include "defs.h"     /* Scott's DL includes  */
#include "spec.h"     /* Scott's DL includes  */

/*********************************************************************
 * resolver.cc -- by Edward Vopata
 *********************************************************************
 *  The deadlock resolver process.
 *  1. Create a list of all the branches in the simulation
 *  2. Wait for deadlock to be reported
 *  3. For each branch int the simulation
 *     a. Determine it's state.
 *        If the branch is sending, intercept the messages and get
 *        the timestamp
 *        If the branch is accepting, query the branch for the
 *        current time.
 *     b. send a Resolver NULL message on every unused outgoing line
 *        of the branch
 *********************************************************************
 */

/*********************************************************************
 * Function:
 *   setup_graph()
 * Parameter:
 *   Param - resolver parameter table
 * Summary:
 *   Create a list of all the branches in the simulation.
 *********************************************************************
 */
static void setup_graph(Param)
struct res_param_rec *Param;
{
    struct list_rec *list;  /* The list of branches */
    register i;             /* Index                */

    Param->list = NULL;  /* Empty List */

       /* Step through the process table (PS) */
    for (i=0; i < Param->PS.num_nodes; i++) {

         /* If the logical process is a branch, create a new
          * list element and add the element to the list
          */
      if (Param->PS.type[i] == BRANCH) {
        if (Param->list == NULL) {
             /* If the list was empty before */
          Param->list =
            (struct list_rec *)malloc (sizeof (struct list_rec));
          list = Param->list;
        }
```

```
        else {
             /* If the list was NOT empty before */
          list->Next =
            (struct list_rec *)malloc (sizeof (struct list_rec));
          list = list->Next;
        }
           /* Put in some extra useful information */
        list->ID   = i;
        list->Brn_pid = Param->PS.ps_pid[i].Branch_pid;
        list->Next = NULL;
      }
    }
}


/*********************************************************************
 * The Resolver Process.
 *********************************************************************
 */

process body resolver()
{
    register i,id,t_id;              /* Indexes                      */
    struct   res_param_rec *Param; /* Resolver Parameter table */
    struct   list_rec *list;        /* List of branches             */
    struct   list_rec state;        /* Past state information       */
    ITEM     Item;                  /* Resolver NULL message        */
    int      Done = FALSE;          /* Termination Flag             */
    struct {
        RES_SEND Res_send;  /* Transaction Pointer to Dest */
        long     pid;       /* Process ID of Destination   */
    } brn_line;

       /* Allocate space for the parameter table, it is large */
    Item.data[0] = '\0';
    Param = (struct res_param_rec *)
        malloc(sizeof(struct res_param_rec));

      /* Get the resolvers Process id */
    Param->my_pid = (process resolver)c_mypid();


      /* Get the parameter table, takes three transactions */
    accept setup1(ID,ID2,dead) {
        Param->ID = ID;
        Param->ID2 = ID2;
        Param->dead = dead; };    /* pid of deadlock detector  */
    accept setup2(PS) { Param->PS = PS; };    /* process table */
    accept setup3(Out) {Param->out = Out; };  /* Out Graph     */

    if (Param->PS.debug&DEBUG_RESOLVE) /* Resolver tracing */
        printf("Resolver: Setup Complete\n");

      /* get the list of branches */
    setup_graph(Param);
```

```
for (;;) {
  select {
    (!Done):  /* wait for a deadlock report */
      accept report(pid) {
        if (Param->PS.debug&DEBUG_RESOLVE) /* Resolver tracing */
                printf("Resolver:Deadlock reported\n");
        treturn 0;
        };

           /* Once deadlock is reported, Resolver it */
        list = Param->list;
        if (Param->PS.debug&DEBUG_RESOLVE) /* Resolver tracing */
            printf("Resolver: list %X\n",list);

            /* For each branch in the list */
        while (list != NULL) {
            id = list->ID;
               /* Get the state of the branch */
            state.state =
                Param->dead.getstate(Param->PS.ps_pid[id].pid);

               /* If in a RUNNING State, abort resolution */
            if ((state.state & 0xF8000000) == (3<<27)) break;

              /* If in a SENDING State, intercept the message */
            if ((state.state & 0xF8000000) == (2<<27)) {

                  /*  DO NOT REMOVE THIS SECTION OF CODE */
               if (Param->PS.debug&DEBUG_RESOLVE)
                 printf("Resolver: Intercept, form %X  to  %X\n",
                    Param->PS.ps_pid[id].pid,
                    state.state & 0x07FFFFFF);
                Param->dead.intercept(Param->PS.ps_pid[id].pid,
                    (state.state & 0x07FFFFFF),
                    Param->my_pid.intercept_response);

                accept intercept_response(item,status) {
                     if (status == 0) {
                        bcopy(item.outbuf,&Item,sizeof(ITEM));
                        state.sim_time = Item.Time_Stamp;
                    }
                    else {
                        fprintf(stderr,"Could Not intercept\n");
                    }
                }
            }
            else  {
                  /* If state is ACCEPTING, query the branch */
               state.sim_time =
                   Param->PS.ps_pid[id].Branch_pid.get_time();
            }

              /* Create a Resolver NULL message, using the
               * time obtained from the intercepted message
               * or from the branch itself
               */
```

```
Item.Item_type  = MSG_NULL;
Item.Time_Stamp = Item.Arrive_Time = state.sim_time;
Item.Hop_Count  = Param->PS.Max_Hop_Count;
Item.Id_of_Src  = Param->ID;
Item.data[0]    = '\0';

   /* For each outgoing line of the branch */
for (i=0;i < Param->out.out_graph[id].num_out; i++) {

  t_id = Param->out.out_graph[id].out[i];

   /* Get a transaction pointer of the destination */
  switch (Param->PS.type[t_id]) {
  case BRANCH:
      brn_line.pid =
        (long)Param->PS.ps_pid[t_id].Branch_pid;
      brn_line.Res_send =
        Param->PS.ps_pid[t_id].Branch_pid.Res_send;
      break;
  case SINK:
      brn_line.pid =
        (long)Param->PS.ps_pid[t_id].Sink_pid;
      brn_line.Res_send =
        Param->PS.ps_pid[t_id].Sink_pid.Res_send;
      break;
  case QUE_SRV:
      brn_line.pid =
        (long)Param->PS.Que_pid[t_id];
      brn_line.Res_send =
        Param->PS.Que_pid[t_id].Res_send;
      break;
  default:
      fprintf(stderr,"Resolver: Invalid type (%d)\n",
          Param->PS.type[t_id]);
  }

    /* If the branch is sending on that line,
     * ignore it.
     */
  if ((state.state & 0xF8000000) == (2<<27) &&
    ((state.state & 0X07FFFFFF) == brn_line.pid)) {
      continue;
  }
     /* otherwise send the resolver NULL message
      * as an asynchronous message
      */
  (*brn_line.Res_send)(Item);
}
   /* Record current State information */
state.ID   = list->ID;
state.Next = list->Next;
if (Param->PS.debug&DEBUG_RESOLVE) {
   printf("old state = %X    new state = %X\n",
       list->state,state.state);
   printf("old sim_time = %lf  new sim_time = %lf\n",
       list->sim_time,state.sim_time);
```

```
            }
            (*list) = state;
            list = list-> Next;
        }
           /* Tell deadlock detector, that deadlock has been
            * resolved.
            */
        Param->dead.enable_detect();

        if (Param->PS.debug&DEBUG_RESOLVE)
          printf("Resolver: RESOLUTION COMPLETE\n");
        fflush(stdout);
        fflush(stderr);

   or
    (Done):  /* During termination, ignore deadlock reports */
     accept report(pid) {
        if (Param->PS.debug&DEBUG_RESOLVE)
           printf("Resolver:Deadlock reported:Ignored\n");
        treturn 0;
        };

   or
         /* accept a terminate message */
      accept term() {};
         Param->dead.term();
         Done = TRUE;
   or
    (Done):
     terminate;   /* Terminate */
   }
   c_sch();    /* Schedule another process */
   delay 1.0;  /* wait a while              */
 }
}
```

F.28: set.cc

```
#include "dclr.h"

/*****************************************************************
 * set.cc -- by Edward Vopata
 *
 * Set operations:
 *  A Set is composed of an array of long integers (16 bits).
 *  The sets are manipulated by bit operations.  The size of the SET
 *  is determined by the number of CHUNKS
 *****************************************************************
 */

/*****************************************************************
 * Function:
 *   set_add()
 * Parameter:
 *   set  -  The Set
 *   elem -  The Element to be added to the SET
 * Summary:
 *    Set the bit in the set corresponding to elem
 *****************************************************************
 */
void set_add (set,elem)
SET set;      /* The SET                   */
int elem;     /* Element to add to the SET */
{
    register shift, entry;  /* Shift and entry variables */

        /* Make sure elem is in proper range */
    if ((elem < 0) || (elem > SETLEN - 1))
        fprintf(stderr, "add_set: element (%d) out of range\n",elem);
    else {
        shift = elem % 32;  /* Calculate which bit in a entry */
        entry = elem / 32;  /* Calculate which entry          */

              /* Set the proper bit in the set */
        set [entry] |= (long)(1 << shift);
        /* printf("set_add:set %x  set[0] %x\n",set,set[0]); */
    }
}

/*****************************************************************
 * Function:
 *   set_union()
 * Parameter:
 *   set1 - A Set
 *   set2 - Another Set
 * Summary:
 *   create the union of two sets.  set1 <- set1 UNION set2
 *
 *****************************************************************
 */
void set_union (set1,set2)
```

```
SET set1, set2;
{
    register i;  /* Index */

      /* Do the union by setting every bit in set1 that is set
       * also set in set2.  Use bitwise OR.
       */
    for (i=0; i < CHUNKS; i++) set1[i] |= set2[i];
}

/***********************************************************************
 * Function:
 *    set_in()
 * Parameter:
 *     set  - A set
 *     elem - An element
 * Summary:
 *    Determine if the element "elem" is in the set.
 *     Return: 0 (FALSE)                 if "elem" is NOT in the set.
 *             1 << (elem % 32) (TRUE) if "elem" is in the set.
 ***********************************************************************
 */
int set_in(set,elem)
SET set;   /* the Set                 */
int elem;  /* the element in question */
{
    register shift, entry;  /* Shift and Entry */

       /* Make sure "elem" is in the proper range */
    if ((elem < 0) || (elem > SETLEN - 1))
        fprintf(stderr, "in_set: element (%d) out of range\n",elem);

    shift = elem % 32;  /* Offset from the start of a chunk */
    entry = elem / 32;  /* Determine which chunk            */
      /* Determine if elem is in the set */
    return (set[entry] & (1 << shift));  /* Return  0 or NOT 0 */
}

/***********************************************************************
 * Function:
 *    set_full()
 * Parameter:
 *    set - A Set
 *    max - Number of bits to check
 * Summary:
 *    Determine if there are "max" number of bits set in the Set.
 *    Returns:  0 (FALSE)    if there are not "max" bits set
 *              1 (TRUE) if there are "max" bits set
 ***********************************************************************
 */
int set_full(set,max)
SET set;
int max;
{
    register i, shift, entry;  /* Shift and Entry */
```

```
    /* Make sure "elem" is in the proper range */
if ((max < 0) || (max > SETLEN))
    fprintf(stderr, "full_set: max (%d) out of range\n",max);

shift = max % 32;  /* Offset from the chunk  */
entry = max / 32;  /* determine proper chunk */

   /* determine if all the bits in the chunks less then
    * entry are set.  If not return FALSE.
    */
for (i=0; i < entry; i++) {
    if ((set [i] & 0xFFFFFFFF) != 0xFFFFFFFF) return FALSE;
}

   /* Check the last (entry) chunk (which may only be a
    * partial chunk) to determine if "shift" number of
    * bits are set.
    */
if ((set [entry] & (long)((1 << shift) - 1)) !=
  (long)((1 << shift) - 1))
    return FALSE;  /* If not return FALSE */

    /* If we get here then there were "max" bits set
     * so return TRUE
     */
return TRUE;
}
```

F.29: stats.cc

```
#include "dclr.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 -1 (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 = -1.0;  /* max. value <= -1 (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 maximum 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  %lf\n",
        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);
    }
}
```

F.30: sock.cc

```
#include "dclr.h"

/*********************************************************************
 * sock.cc -- by Edward Vopata
 *
 * sock.cc -- handle socket or file interaction.
 *      getline    -- read a line from a socket or file.
 *      putline    -- write a line to a socket of file.
 *  These functions handle the input and output to the socket or file
 *  as necessary. Each function has 3 parameters (sock, line, debug).
 *  sock  : is the socket descriptor or file descriptor.  getline and
 *          putline assume that the socket or file is correctly open.
 *  line  : is the the input/output string, it is 128 character long.
 *  debug : is a flag to enable debugging. In debug mode getline and
 *          putline will print what was read on the socket/file in
 *          getline, and what WAS written on the socket/file after
 *          the write was completed.
 *********************************************************************
 */

/*********************************************************************
 * Function:
 *   getline()
 * Parameter:
 *     sock  - socket/file descriptor
 *     line  - input buffer
 *     debug - Debugging flag
 * Summary:
 *   Read a line of data from the socket into the buffer "line".
 * Description
 *   1. Clear the input buffer. Zero file "line" (bzero).
 *   2. Read from the sock (descriptor) into line (size MAXLINE).
 *   3. Make sure the line ends with a null (\0), since the line will
 *      be treated as a string.
 *********************************************************************
 */
void getline(sock,line,debug)
int sock;           /* Socket/file descriptor */
char *line;         /* I/O buffer             */
int debug;          /* Debugging flag         */
{
       /* Clear the buffer (to zero) */
    bzero(line,(sizeof(char)*MAXLINE));
       /* Read the socket/file into the buffer */
    read(sock,line,MAXLINE);
       /* Put a Null (\0) at the end of the buffer */
    line[MAXLINE-1] = '\0';
       /* If Debug Mode then Print the original line and flush
        * standard out to make sure the line is displayed.  */
    if (debug) {
        printf("getline:[%s]\n", line);
        fflush(stdout);
    }
```

```
}

/*********************************************************************
 * Function:
 *    putline()
 * Parameter:
 *      sock  - socket/file descriptor
 *      line  - output buffer
 *      debug - Debugging flag
 * Summary:
 *    Write a line of data from buffer "line" to the socket.
 * Description:
 *    1. Pad the line with spaces on the right to form a line
 *       of length == MAXLINE.
 *    2. Write the padded line to the socket (or file).
 *********************************************************************
 */
void putline(sock,line,debug)
int sock;          /* Socket/file descriptor */
char *line;        /* I/O buffer             */
int debug;         /* Debugging flag         */
{
    char outline[130];  /* outgoing line */

        /* Pad the outgoing line with blank on the right */
    sprintf(outline,"%-127s\n",line);

       /* If Debug Mode then Print the outgoing line */
    if (debug) printf("putoutline:[%s]\n",outline);
       /* Write the outgoing line to the socket or file */
    write(sock,outline,128);
       /* If Debug Mode then Print the original line and flush
        * standard out to make sure the line is displayed.  */
    if (debug) {
        printf("putline:[%s]\n",line);
        fflush(stdout);
    }
}


/*********************************************************************
 * May want to redo this section using "fgets()" to read in lines
 * and "fprintf()" to write lines.  This will require a "FILE"
 * pointer, which can be create by using "fdopen" on the socket
 * descriptor.
 *    -- Ed Vopata
 *********************************************************************
 */
```

F.31: distrib.cc

```
#include <math.h>
#include "rand.h"
#include "distrib.h"

/*******************************************************************
 * distrib.cc -- by Edward Vopata
 *******************************************************************
 * Stochastic distribution functions.
 *   the get_time() function returns a service or arrival time,
 *   based on the specified stochastic distribution function and
 *   parameters.
 *******************************************************************
 */

/* Forward Declaration of the stochastic distribution functions */
long   binomial();
long   poisson();
long   uniform();
double beta();
double erlang();
double expntl();
double gamma();
double lognormal();
double normal();
double weibull();

/*******************************************************************
 * Function:
 *   get_time()
 * Parameter:
 *   dis - specifies the stochastic distribution function to use,
 *         and the parameter to use with the function.
 * Summary:
 *   Use the specified stochastic distribution function to generate
 *   a random variable. The get_time function is used to generate
 *   service and arrival times.
 * Returns:
 *   a floating point value greater than zero (0). Note: the time
 *   return may be zero (0).
 *******************************************************************
 */
double get_time(dis)
struct distrib_rec dis;
{
  double time;  /* arrival or service time */

  do {
      /* Determine which distribution function to use */
    switch(dis.type) {

      /* If FIXED, time <= the fixed.time parameter */
    case FIXED:
      time = dis.DIS.fixed.time;
```

```
  if (dis.debug) printf("Fixed time = %lf\n",time);
  break;

  /* If UNIFORM, time <= uniform() */
case UNIFORM:
  time = uniform(dis.DIS.uniform.lower,dis.DIS.uniform.upper);
  if (dis.debug) printf("Uniform time = %lf\n",time);
  break;

  /* If POISSON, time <= poisson() */
case POISSON:
  time = poisson(dis.DIS.poisson.mean);
  if (dis.debug) printf("Poisson time = %lf\n",time);
  break;

  /* If BINOMIAL, time <= binomial() */
case BINOMIAL:
  time = binomial(dis.DIS.binomial.trials,dis.DIS.binomial.prob);
  if (dis.debug) printf("Binomial time = %lf\n",time);
  break;

  /* If EXPNTL, time <= expntl() */
case EXPNTL:
  time = expntl(dis.DIS.expntl.mean);
  if (dis.debug) printf("Expntl time = %lf\n",time);
  break;

  /* If NORMAL, time <= normal() */
case NORMAL:
  time = normal(dis.DIS.normal.mean,dis.DIS.normal.stdev);
  if (dis.debug) printf("Normal time = %lf\n",time);
  break;

  /* If GAMMA, time <= gamma() */
case GAMMA:
  time = gamma(dis.DIS.gamma.mean,dis.DIS.gamma.k);
  if (dis.debug) printf("Gamma time = %lf\n",time);
  break;

  /* If BETA, time <= beta() */
case BETA:
  time = beta(dis.DIS.beta.k1,dis.DIS.beta.k2);
  if (dis.debug) printf("Beta time = %lf\n",time);
  break;

  /* If ERLANG, time <= erlang() */
case ERLANG:
  time = erlang(dis.DIS.erlang.mean,dis.DIS.erlang.k);
  if (dis.debug) printf("Erlang time = %lf\n",time);
  break;

  /* If LOGNORMAL, time <= lognormal() */
case LOGNORMAL:
  time=lognormal(dis.DIS.lognormal.mean,dis.DIS.lognormal.stdev);
  if (dis.debug) printf("Lognormal time = %lf\n",time);
  break;
```

```
        /* If WEIBULL, time <= weibull() */
      case WEIBULL:
        time = weibull(dis.DIS.weibull.shape,dis.DIS.weibull.scale);
        if (dis.debug) printf("Weibull time = %lf\n",time);
        break;
      }
      /* Keep doing this until a positive time value is found
       * (This may loop forever in some cases?
       */
    } while (time < 0.0);

      /*  Truncate the functions if min or max time is set (> 0) */
    if (dis.min_time > 0.0 && time < dis.min_time) time = dis.min_time;
    if (dis.max_time > 0.0 && time > dis.max_time) time = dis.max_time;
    return time;  /* return the 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 *= drand01();
    } 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 = drand01()) == 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 (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)));
}
```

F.32: stats_rpt.cc

```
#include "dclr.h"

/********************************************************************
 * stats_report.cc -- by Edward Vopata
 ********************************************************************
 * function Print_Stats() -- create a file of the collective
 *   statistical reports.  This is a more complete report then what
 *   is sent to the graphics front-end.
 ********************************************************************
 */

   /* Statistics Report Status */
#define NOT_DONE  0
#define TERM_TIME 1
#define ALL_SUNK  2

/********************************************************************
 * Function:
 *   Print_Stats()
 * Parameter:
 *   Param  - collector parameter table
 *   head   - pointer to the head of the
 *   status - status of the collector
 * Summary:
 *   add a collective statistics report to the "Stats_Report" File
 * Description:
 *   open the file "STATS_REPORT_NAME" (defined in dclr.h) and
 *   write the current statistics report to the file.  This report
 *   is in a human readable format.
 ********************************************************************
 */
void Print_Stats(Param,head,status)
struct col_param *Param;
struct col_rec *head;
int    status;
{
    FILE     *fp;   /* File pointer  */
    register i;     /* Index         */
    double   Time;  /* Interval Time */

      /* Open the file */
    fp = fopen (STATS_REPORT_NAME,"a");

    Time = (head->i_num * Param->PS.stats_interval) +
        Param->PS.stats_interval;

      /* Print header, with interval number and interval time */
    fprintf(fp,"\n");
    fprintf(fp,"==============================");
    fprintf(fp,"==============================\n");
    fprintf(fp,"==  Interval Number: %-4ld  ", head->i_num);
    fprintf(fp,"Interval Time: %-14.4lf ==\n", Time);
    fprintf(fp,"==============================");
```

```
fprintf(fp,"==============================\n");

  /*  Print Status of Collector */
switch (status) {
case TERM_TIME:
  fprintf(fp,"Status: Termination Time reached\n");
  break;
case ALL_SUNK:
  fprintf(fp,"Status: All Sunk\n");
  break;

case NOT_DONE:
default:
  fprintf(fp,"Status: Normal\n");
  break;
}

   /* For each logical process that has reported statistics
    * print the stats for that lp.
    */
for (i=0; i < Param->PS.num_nodes; i++) {
  if (set_in(head->modified,i)) {
    switch (Param->PS.type[i]) {

    case SOURCE:
        /* Print Source Statistics */
     fprintf(fp,"\n");
     fprintf(fp,"Source: %2d  Interval: %.4lf  ",i,Time);
     fprintf(fp,"Simulation Time: %.4lf\n",
        head->Stats[i].Src_stats->sim_time);
     fprintf(fp,"        Inter Arrival Time             Number\n");
     fprintf(fp,"     Ave        STD        MAX          Left\n");
     fprintf(fp,"%8.3lf   %8.3lf   %8.3lf        %4ld\n",
        head->Stats[i].Src_stats->ave_iit,
        head->Stats[i].Src_stats->std_iit,
        head->Stats[i].Src_stats->max_iit,
        head->Stats[i].Src_stats->num_left);
     break;

    case QUE_SRV:
        /* Print Queue and Server statistics */
     fprintf(fp,"\n");
     fprintf(fp,"Queue/Server: %2d  Interval: %.4lf  ",i,Time);
     fprintf(fp,"Simulation Time: %.4lf\n",
        head->Stats[i].Q_Srv_stats->sim_time);
     fprintf(fp,"Queue:  Full: %6.2lf%%  Num Through Queue: %ld",
        head->Stats[i].Q_Srv_stats->q_stats.per_full,
        head->Stats[i].Q_Srv_stats->q_stats.num_through_q);
     fprintf(fp,"\n");
     fprintf(fp,"      Average Time in Queue           ");
     fprintf(fp,"Average in Queue     Num in\n");
     fprintf(fp,"     Ave        STD        MAX        Ave        ");
     fprintf(fp,"STD        MAX Queue\n");
     fprintf(fp,"%8.3lf  %8.3lf  %8.3lf  ",
        head->Stats[i].Q_Srv_stats->q_stats.ave_time_in_q,
        head->Stats[i].Q_Srv_stats->q_stats.std_time_in_q,
```

```
          head->Stats[i].Q_Srv_stats->q_stats.max_time_in_q);
        fprintf(fp,"%8.3lf  %8.3lf  %8.3lf  %4ld\n",
          head->Stats[i].Q_Srv_stats->q_stats.ave_in_q,
          head->Stats[i].Q_Srv_stats->q_stats.std_in_q,
          head->Stats[i].Q_Srv_stats->q_stats.max_in_q,
          head->Stats[i].Q_Srv_stats->q_stats.num_in_q);
        fprintf(fp,"\n");

          /* Print the Server Stats */
        fprintf(fp,"Server:   Busy: %6.2lf%%\n",
          head->Stats[i].Q_Srv_stats->per_busy);
        fprintf(fp,"      Average Time In System        ");
        fprintf(fp,"Average Service Time     Number\n");
        fprintf(fp,"     Ave       STD       MAX       ");
        fprintf(fp,"Ave       STD       MAX  Serviced\n");
        fprintf(fp,"%8.3lf  %8.3lf  %8.3lf  ",
          head->Stats[i].Q_Srv_stats->ave_time_in_sys,
          head->Stats[i].Q_Srv_stats->std_time_in_sys,
          head->Stats[i].Q_Srv_stats->max_time_in_sys);
        fprintf(fp,"%8.3lf  %8.3lf  %8.3lf    %4ld\n",
          head->Stats[i].Q_Srv_stats->ave_serve_time,
          head->Stats[i].Q_Srv_stats->std_serve_time,
          head->Stats[i].Q_Srv_stats->max_serve_time,
          head->Stats[i].Q_Srv_stats->num_served);
        break;

      case SINK:
          /* Print Sink Stats */
        fprintf(fp,"\n");
        fprintf(fp,"Sink: %2d  Interval: %.4lf  ",i,Time);
        fprintf(fp,"Simulation Time: %.4lf\n",
          head->Stats[i].Sink_stats->sim_time);
        fprintf(fp,"  Number Sunk:  %ld\n",
          head->Stats[i].Sink_stats->num_sunk);
        break;

      default:
        fprintf(stderr,"Print_Stats: Invalid type (%d)\n",
          Param->PS.type[i]);
        break;
      }
    }
  }
  fclose(fp);  /* Don't forget to close the file */
}
```

F.33: out_graph.cc

```
#include "dclr.h"
#include "graph.h"
#include "Pid.h"
#include "defs.h"
#include "spec.h"

/*******************************************************************
 * out_graph.cc -- by Edward Vopata
 *******************************************************************
 * function : print_out_graph()  creates a file "OUT_GRAPH_NAME" as
 *  defined in dclr.h containing the Out Graph of the input model.
 *  The Out Graph is a list of the logical process and the process
 *  which they may send messages.  The Out Graph is very useful in
 *  insuring that the input model was correctly received.
 *******************************************************************
 */

/*******************************************************************
 * Function:
 *     print_out_graph()
 * Parameter:
 *    PS     : Process table
 *    Out    : Out Graph
 *    Res    : List of resolver processes
 *    num_VP : Number of resolver processes
 * Summary:
 *   Print an Out Graph of the logical process and who they can
 *   send messages to.
 * Description:
 *   open the file "OUT_GRAPH_NAME" (defined in dclr.h).
 *   for each logical process write it's process id, the number of
 *   incoming line, the number of outgoing lines, and the ID of any
 *   logical process that the process can send messages (if
 *   appropriate).  Also print the process id of the collector,
 *   terminator and all the resolver processes.
 *******************************************************************
 */
void print_out_graph(PS,Out,Res,num_VP)
struct PS_REC *PS;
struct out_list *Out;
process Resolver *Res;
int num_VP;
{
  FILE *fp;         /* File pointer */
  register i,j;     /* Indexes      */

     /* Open the file */
  fp = fopen(OUT_GRAPH_NAME,"w");

     /* For each logical process in the process table (PS) */
  for (i=0;i < PS->num_nodes; i++) {
     /* For each type of logical process */
    switch (PS->type[i]) {
```

```
    case SOURCE:
          /* For the Source: print ID, pid, and ID of destination */
       fprintf(fp,"Source [%3d] pid[%X]  To[%3d]\n",i,
          PS->ps_pid[i].pid, Out->out_graph[i].out[0]);
       break;

    case SINK:
          /* For the Sink: print ID, pid, number incoming liens */
       fprintf(fp,"Sink   [%3d] pid[%X] #in[%3d]\n",i,
          PS->ps_pid[i].pid, Out->out_graph[i].num_in);
       break;

    case QUE_SRV:
          /* For the Server: print ID, pid, ID of destination */
       fprintf(fp,"Server [%3d] pid[%X]  To[%3d]\n",i,
          PS->ps_pid[i].pid, Out->out_graph[i].out[0]);
         /* For the Queue: print ID, pid, number of incoming lines */
       fprintf(fp,"Queue  [%3d] pid[%X] #in[%3d]\n",i,
          PS->Que_pid[i], Out->out_graph[i].num_in);
       break;

    case BRANCH:
          /* For the Branch: print ID, pid, number incoming lines,
           * number of outgoing lines, and the list of destinations
           */
       fprintf(fp,"Branch [%3d] pid[%X] #in[%3d] #out[%3d] To ",i,
          PS->ps_pid[i].pid, Out->out_graph[i].num_in,
          Out->out_graph[i].num_out);
       for (j=0;j < Out->out_graph[i].num_out; j++)
         fprintf(fp,"[%3d] ",Out->out_graph[i].out[j]);
       fprintf(fp,"\n");
       break;
    }
  }

    /* Print pid of Collector, Terminator and every Resolver */
  fprintf(fp,"Collector    pid[%X]\n",PS->Col_pid);
  fprintf(fp,"Terminator   pid[%X]\n",PS->Term_pid);
  for (i=0;i<num_VP; i++)
    fprintf(fp,"Res    [%3d] pid[%X]\n",i,Res[i]);

  fclose(fp);  /* Don't forget to close the file */
}
```

F.34: sizeof.cc

```
#include "dclr.h"
#include "graph.h"
#include "Pid.h"
#include "defs.h"
#include "queue.h"

/*********************************************************************
 * sizeof.cc -- by Edward Vopata
 *********************************************************************
 * function: print_size()
 *  Creates a file of name "SIZE_OF_NAME" defined in dclr.h. The
 *  generic name is "Size_Of". This function prints the sizes of
 *  various structures used by the simulator. The Size_Of report is
 *  very useful in finding structure larger than 4K. (The AT&T C
 *  compiler (Ver 4.1) has a bug in which when passing structures
 *  of larger then 4K cause the program to hang or dump core).
 *  Scott Hammond and I found the previously unknown bug.  AT&T
 *  techincal support certified that the bug was original, and will
 *  be corrected in the Ver 4.3 update.
 *********************************************************************
 */

/*********************************************************************
 * Function:
 *   print_size()
 * Summary:
 *   print the sizes of various structures used by the distributed
 *   simulator
 * Description:
 *   open the file "SIZE_OF_NAME" (defined in dclr.h), print the
 *   size of the various structures (using the C sizeof() function).
 *********************************************************************
 */
void print_size()
{
  FILE *fp;  /* File pointer */

  fp = fopen(SIZE_OF_NAME,"w");  /* Open the file */

      /* Standard types like int, char, double */
  fprintf(fp,"\nSize of Standard Types:\n");
  fprintf(fp,"int                         %d\n",sizeof(int));
  fprintf(fp,"long                        %d\n",sizeof(long));
  fprintf(fp,"double                      %d\n",sizeof(double));
  fprintf(fp,"char                        %d\n",sizeof(char));

      /* Values used by the simulator */
  fprintf(fp,"\nValues:\n");
  fprintf(fp,"MAXPROC                     %d\n",MAXPROC);
  fprintf(fp,"MAXFAN                      %d\n",MAXFAN);
  fprintf(fp,"D_NPROCS                    %d\n",D_NPROCS);
  fprintf(fp,"D_NPORS                     %d\n",D_NPORS);
```

```
    /* Size of Stochastic Distribution Function Structures */
fprintf(fp,"\nSize of Stochastic Distributions Structures:\n");
fprintf(fp,"struct fixed_rec      %d\n",sizeof(struct fixed_rec));
fprintf(fp,"struct uniform_rec    %d\n",sizeof(struct uniform_rec));
fprintf(fp,"struct poisson_rec    %d\n",sizeof(struct poisson_rec));
fprintf(fp,"struct binomial_rec   %d\n",
  sizeof(struct binomial_rec));
fprintf(fp,"struct expntl_rec     %d\n",sizeof(struct expntl_rec));
fprintf(fp,"struct normal_rec     %d\n",sizeof(struct normal_rec));
fprintf(fp,"struct gamma_rec      %d\n",sizeof(struct gamma_rec));
fprintf(fp,"struct beta_rec       %d\n",sizeof(struct beta_rec));
fprintf(fp,"struct erlang_rec     %d\n",sizeof(struct erlang_rec));
fprintf(fp,"struct lognormal_rec %d\n",
    sizeof(struct lognormal_rec));
fprintf(fp,"struct weibull_rec    %d\n",sizeof(struct weibull_rec));
fprintf(fp,"struct distrib_rec    %d\n",sizeof(struct distrib_rec));
    /* Parameter for the various logical processes */
fprintf(fp,"\nSize of Parameters:\n");
fprintf(fp,"struct src_param      %d\n",sizeof(struct src_param));
fprintf(fp,"struct sink_param     %d\n",sizeof(struct sink_param));
fprintf(fp,"struct srv_param      %d\n",sizeof(struct srv_param));
fprintf(fp,"struct q_param        %d\n",sizeof(struct q_param));
fprintf(fp,"struct brn_param      %d\n",sizeof(struct brn_param));
fprintf(fp,"struct col_param      %d\n",sizeof(struct col_param));
fprintf(fp,"struct term_param     %d\n",sizeof(struct term_param));
fprintf(fp,"struct res_param_rec %d\n",
    sizeof(struct res_param_rec));

    /* Process Table and Processes Data Sizes */
fprintf(fp,"\nSize of Process Table & Data Structures:\n");
fprintf(fp,"struct PS_REC          %d\n",sizeof(struct PS_REC));
fprintf(fp,"union  PS_Data_Rec     %d\n",sizeof(union PS_Data_Rec));
fprintf(fp,"PS_Data_Rec * MAXPROC %d\n",
    sizeof(union PS_Data_Rec)*MAXPROC);
fprintf(fp,"struct Src_rec         %d\n",sizeof(struct Src_rec));
fprintf(fp,"struct Q_Srv_rec       %d\n",sizeof(struct Q_Srv_rec));
fprintf(fp,"struct Sink_rec        %d\n",sizeof(struct Sink_rec));
fprintf(fp,"struct fan_rec         %d\n",sizeof(struct fan_rec));
fprintf(fp,"struct Branch_rec      %d\n",sizeof(struct Branch_rec));
fprintf(fp,"struct queue_list_rec %d\n",
    sizeof(struct queue_list_rec));

    /* Size of Statistics Collection Structures */
fprintf(fp,"\nSize of Statistics Structures:\n");
fprintf(fp,"struct src_stats_rec   %d\n",
    sizeof(struct src_stats_rec));
fprintf(fp,"struct q_stats_rec     %d\n",
    sizeof(struct q_stats_rec));
fprintf(fp,"struct q_srv_stats_rec %d\n",
    sizeof(struct q_srv_stats_rec));
fprintf(fp,"struct sink_stats_rec  %d\n",
    sizeof(struct sink_stats_rec));
fprintf(fp,"struct col_rec         %d\n",sizeof(struct col_rec));
fprintf(fp,"struct stats_rec       %d\n",sizeof(struct stats_rec));
fprintf(fp,"SET                    %d\n",sizeof(SET));
```

```
    /* Size of Messages: (Item and Null) */
  fprintf(fp,"\nSize of Message:\n");
  fprintf(fp,"struct Item_rec        %d\n",sizeof(struct Item_rec));

    /* Size of Miscellaneous Structures */
  fprintf(fp,"\nOther structures:\n");
  fprintf(fp,"struct out_rec         %d\n",sizeof(struct out_rec));
  fprintf(fp,"struct out_list        %d\n",sizeof(struct out_list));
  fprintf(fp,"struct list_rec        %d\n",sizeof(struct list_rec));
  fprintf(fp,"CALLERS                %d\n",sizeof(CALLERS));
  fprintf(fp,"OUTBUF                 %d\n",sizeof(CALLERS));

  fclose(fp); /* Don't forget to close the file */
}
```

F.35: connect.c

```
/*****************************************************************
 * connect.c -- by Edward Vopata
 *****************************************************************
 *  This is the startup program.  This program opens a socket
 *  connection and "exec's" the Distributed simulator.  This
 *  program passes the distributed simulator the socket descriptor,
 *  and the arguments that the startup program was started with.
 *  This program was designed to run on a AT&T 3b2-400 running
 *  WIN/3B TCP/IP.
 *****************************************************************
 */
#include <stdio.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <netdb.h>

#define DEBUG_DOCNT 13   /* Debugging switch of socket connection */
#define MAX_DEBUG   32   /* Maximum number of debugging switches  */

int  debug = 0;  /* debug flag */
   /* Max number of arguments + NULL + socket number */
char *argo[MAX_DEBUG + 2];   /* Argument list */
char name_str[50];   /* Path to the distributed simulator */
char sock_str[10];   /* socket descriptor string          */

main(argc,argv)
int  argc;
char **argv;
{
    int     i, j;    /* indexes           */
    int    sock;     /* socket descriptor */

       /* create a new command line to be exec'ed */
    for (i=1; i<argc; i++) {
        argo[i+1] = argv[i];
            /* if DEBUG_DOCNT switch is on set debug flag */
        if (atoi (argv[i]) == DEBUG_DOCNT) debug = 1;
    }
    argo[argc+1] = argv[argc++];

      /* open the socket */
    do_connect(&sock);

    strcpy(name_str,SIMULATION); /* Put Program name in argo[0]    */
    argo[0] = name_str;          /* Put -<socket_no> in argo[1]    */

        /* put socket id into a string, prefaced with a '-' */
        /* make sure to remove the '-' when you read it     */
        /* use '-' to id the socket number                  */
    sprintf(sock_str,"-%d",sock);
    argo[1] = sock_str;      /* Put -<socket_no> in argo[1] */
```

```
    if (debug) printf("argc %d\n",argc); /* DOCNT tracing */

       /* Put argv[1..argc-1] -> argo[2..argc] */
    for (i=0; i <= argc; i++)
        if (debug) printf ("argo[%d] = '%s'\n",i, argo[i]);

      /* Exec the distributed simulator */
    execv(SIMULATION,argo);
    printf("EXEC Failed\n"); /* This should never be printed */
}

/*********************************************************************
 * Function:
 *   do_connect()
 * Parameters:
 *   sock - socket descriptor
 * Summary:
 *   TCP/IP socket server.  Waits until someone tries to connect to
 *   the "xeroxsim" socket port. Then open a reliable STREAM socket.
 *********************************************************************
 */
do_connect(sock)
int *sock;
{
    struct sockaddr_in sin;
    struct sockaddr *addr;
    struct servent *sp;
    int    s, *addrlen;
    int    accept();

       /* Get tcp/ip service "xeroxsim" */
    sp = getservbyname("xeroxsim", "tcp");
    if (sp == NULL) {
        fprintf(stderr, "xeroxsim: Couldn't find service\n");
        exit(1);
    }
    bzero ((char *) &sin, sizeof(sin));

       /* Open a socket port */
    if (debug) printf("xeroxsim: getting socket\n");
    if ((s = socket (AF_INET, SOCK_STREAM, 0)) < 0) {
        if (debug) printf("socket failed\n");
        exit(1);
    }
    sin.sin_family = AF_INET;
    sin.sin_addr.s_addr = htonl (INADDR_ANY);
    sin.sin_port = sp->s_port;

       /* bind the socket to the "xeroxsim" service */
    if (debug) printf("xeroxsim: attempting bind\n");
    if (bind (s, (struct sockaddr *) &sin, sizeof (sin)) < 0) {
        perror("xeroxsim: bind failed");
        exit(1);
    }

      /* Wait until someone tries to connect to the service */
```

```
    if (debug) printf("xeroxsim: listening\n");
    listen (s, 1);

      /* When they connect, accept the connection and get a
       * socket descriptor for the connection.
       * (the accept() function conflicts with the Concurrent C
       *  accept statement, so this program must be a regular C
       *  program, that exec's the Concurrent C distributed
       *  simulator).
       */
    if (debug) printf("xeroxsim: accept\n");
    (*sock) = accept (s, addr, addrlen);

      /* Do_connect tracing */
    if (debug) printf("sock = %d\n",*sock);
    if (debug) printf("xeroxsim: accepted\n");

        /* Since we are exiting, tell it no more data is to be
         * read or sent.
         */
    shutdown (s, 2);
    close(s);  /* Close the service */
}
```

F.36: Xerox.c

```
/*****************************************************************
 * Xerox.c -- By Edward Vopata
 *
 * Xerox.c is a program used to test TCP/IP socket communication.
 * This program simulates the socket operation of the graphics
 * front-end.  The program first opens a socket connection to the
 * distributed simulator.  If this fails the program exits, otherwise
 * the program will prompt the user for a file name. This file will
 * contain the input model.  The program will read the input model
 * and send it to the distributed simulator.  The program will then
 * wait until the distributed simulator send stats reports back.
 * These stats reports will be displayed, and the user can either
 * cause a continue or a terminate simulation control message to be
 * sent back. (This program runs only on a 3B2 with WIN/3B2 TCP/IP).
 *****************************************************************
 */

#include <stdio.h>
#include "/usr/netinclude/sys/types.h"
#include "/usr/netinclude/sys/socket.h"
#include "/usr/netinclude/netinet/in.h"
#include "/usr/netinclude/netdb.h"

    /* The hostname (machine name) where the */
    /* distributed simulator resides         */
#define HOSTNAME "november"

main(argc,argv)
int argc;
char **argv;
{
    int i;                  /* index                        */
    int sock;               /* the socket descriptor        */
    int flag = 0;           /* flag for handling "(end)"    */
    char inline[129];       /* the incoming message         */
    char outline[129];      /* the outgoing message         */
    FILE *f;                /* the input model file         */

      /* Open the connection and get the socket descriptor */
    doconnect(&sock);

      /* Get the file name of the input model and open the file */
    printf("Enter File Name ==>  ");
    fflush(stdout);
    gets(inline);
    printf("\n");
    if((f = fopen(inline,"r")) == NULL)
       {printf("Bad File:'%s'.\n",inline); close(sock); exit(1); }

       /* Send the input model to the distributed simulator */
    bzero(inline,sizeof(inline));
    while (fgets(inline,128,f) != NULL) {
        printf("Sending: |%s|\n",inline);
```

```
          sprintf(outline,"%-128s",inline);
          write(sock,outline,128);
          bzero(inline,sizeof(inline));
      }
      fclose(f);  /* Close the file, we're done with it */

      do {
         do {
            /* Read in the incoming messages (blocking read) */
          bzero(inline,sizeof(inline));
          if (read(sock, inline, 128)==0) break;

             /* if the message is an "(abort)" then ABORT */
          if (inline[1] == 'a') ABORT(sock);
             /* if the message is an "(end)" then set flag */
          if (inline[1] == 'e') flag = 1;

             /* Print the line */
          printf("Recv'd: |%s|\n", inline);

            /* when a "($$)" is encounted exit the loop */
          if (inline[1] == '$' || inline[2] == '$') break;
         } while (1);

           /* If the flag is set then we done, exit the loop */
         if (flag) break;

           /* Prompt the user for either a continue or a terminate */
           /*   if response is "q" then ABORT                         */
           /*      "     "     "a" then send terminate control msg */
           /*                              "((96 0))"                */
           /*    otherwise send a continue control msg               */
           /*                              "((97 0))"                */
         printf("Enter Reply ==>  ");
         fflush(stdout);
         gets(inline);
         switch (inline[0]) {
            case 'q'  : ABORT(sock);
            case 'a'  : sprintf(outline,"%-128s","((96 0))"); break;
            case '\0' :
            default   : sprintf(outline,"%-128s","((97 0))"); break;
         }

            /* Send the Simulation Control Message */
         printf("Sending: |%s|\n",outline);
         fflush(stdout);
         write(sock,outline,128);
      } while (inline[0] != 'a');

      close(sock); /* close the socket */
}


/*
 *  ABORT - close the socket and exit the program
 */
```

```
ABORT(sock)
int sock;
{
  close(sock);  /* close the socket */
  exit(1);
}

/*
 * doconnect() --
 *    returns - socket descriptor in "s".
 *    side effects - may open a socket.
 *    if doconnect cannot open the socket it will exit().
 */

doconnect(s)
int *s;

{
    struct sockaddr_in server;
    struct hostent *hp;
    struct servent *sp;

       /* Get Host Name (See #define above) */
    hp = gethostbyname(HOSTNAME);

      /* If gethostbyname fails, then print error message and exit */
    if (hp == NULL) {
        fprintf(stderr, "Unknown host\n");
        exit(1);
    }

      /* Get TCP/IP server (The distributed simulator) */
    sp = getservbyname("xeroxsim", "tcp");

      /* If the distributed simulator has not been started */
      /* then print error message and exit                 */
    if (sp == NULL) {
        fprintf(stderr, "Can't find server xeroxsim\n");
        exit(1);
    }

      /* TCP/IP Socket setup */
    bzero ((char *) &server, sizeof(server));
    bcopy (hp->h_addr, (char *) &server.sin_addr, hp->h_length);
    server.sin_family = hp->h_addrtype;
    server.sin_port = sp->s_port;

      /* Open a STREAM socket (reliable bidirectional) */
    *s = socket (AF_INET, SOCK_STREAM, 0);

      /* If the open fails, print error message and exit */
    if (s < 0) {
        printf("error creating socket\n");
        exit(1);
    }
```

```
    /* Establish the connection:                                  */
    /* If the connection fails, print error message and exit */
   if (connect (*s, (char *) &server, sizeof(server))< 0) {
      printf("connect failed\n"); exit(1); }

    /* Success: return the socket descriptor */
   printf("connected!!!\n");
}
```

Distributed Discrete-Event Simulation in Concurrent C

by

Edward William Vopata

B. S., Kansas State University, 1986

AN ABSTRACT OF A MASTER'S THESIS

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

In this thesis we describe the implementation of a distributed discrete-event simulator for the distributed simulation of queueing network models. The distributed simulator is written in the distributed programming language Concurrent C [GEHA88] and runs on a loosely coupled multiprocessor computer system. The distributed simulator implements the distributed simulation algorithm proposed in [CHAN81] and uses a deadlock detection and resolution strategy to cope with the problems of deadlock. The deadlock detection and resolution strategy makes use of the distributed deadlock detection mechanism that is implemented as part of the kernel of Concurrent C and described in [HAMM88], and a deadlock resolution mechanism implemented as part of our distributed simulator. We have shown that deadlock detection in the kernel of a distributed programming language and deadlock resolution at the application level is a valid approach to distributed simulation.