SIMULATING A DISTRIBUTED FILE SYSTEM WITH VARIOUS TYPES OF NETWORKS

by

MONTE L. HALL

B. S., Kansas State University, 1986

A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1988

Approved by:

Virg Wallentine
(Major Professor)

LD
2668
.R4
CMSC
1988
H35
C. 2

A11208 207576

Table of Contents

Acknowledgements ii

List of Figures. iv

List of Tables vi

Chapters

1. Purpose of Paper 1

2. Introduction to Simulation
   2.1. Simulation Concepts. 6
   2.2. Simulation Environment 12

3. Verification of the Distributed Concurrent C Simulator
   3.1. Simulation Graphs. 16
   3.2. Results of Verification. 23

4. Distributed File System
   4.1. Modeling as a Queueing Network 27
   4.2. Implementation 38

5. Networks
   5.1. Characteristics of Networks. 48
   5.2. Throughput Derivation. 58

6. Results
   6.1. Results of Simulation Runs 62
   6.2. Validation of Project. 62

7. Conclusions and Future Research. 68

References 72

Appendices
- Appendix A 73
- Appendix B 81
- Appendix C 93
- Appendix D 96
- Appendix E 98
- Appendix F 119
- Appendix G 121
- Appendix H 125
- Appendix I 130
- Appendix J 151

Acknowledgements

I would like to thank my major professor, Virg Wallentine, for his continual support and ideas. His interest in simulation and distributed systems has encouraged me to learn more about these areas and to further pursue my interest in networking. I appreciate the numerous hours of discussion which has kept my motivation level very high.

My gratitude also goes out to my other committee members, Rich McBride (who introduced me to networks), and Bill Hankley for their time in helping me edit this report and serving on my committee.

I would like to thank my fellow simulation group members, Scott Hammond, Ed Vopata, and Jim Butler, without whom I would not have had a simulator to run my implementation. Their ideas and discussions have helped me learn more about simulation and have made working on this project a very rewarding experience. Ed Vopata is specially thanked for his tireless efforts in setting up the new laser printer, and finding just the right nroff command for me to use in the printing of this report.

My appreciation goes out to Li-Fang Hsieh who was always there to lend a sympathetic ear. Your support and encouragement have helped me a great deal. Our friendship shall last a lifetime. Thank you.

I wish to thank my beloved parents who have dedicated their entire lives to educating the young leaders of tomorrow. Their love and support in all of the endeavors in my life has been irreplaceable. I believe that impressing upon me the value of a good education and the rewards that can follow has greatly attributed to my optimistic outlook on life. I am forever grateful for all you have done for me. I love you both very much. Thank you again.

Finally I wish to express my deepest appreciation to my future fiancee and wife, Lori, for her patience and understanding during the months of long days I spent on this project that were not spent with her. You have given me the ultimate motivation to complete this degree. And now that I have finally achieved this goal, I am ready to spend the rest of my life with you. I will always be there for you. I love you very much.

List of Figures

Figure 2.1
- Basic Nodes in a Simulation Graph 8

Figure 2.2
- Nodes of More Complex Simulation Graphs 9

Figure 2.3
- Communication Ring Network. 11

Figure 3.1
- Single-Queue Single-Server System. 17

Figure 3.2
- Single-Queue Single-Server Feedback System 17

Figure 3.3
- Traffic-Merging System 17

Figure 4.1
- Simple Model of a System. 28

Figure 4.2
- Simple Model of a System with Response Time Line 28

Figure 4.3
- A Model of a Host with a Delay Server 30

Figure 4.4
- A Model of a Host 32

Figure 4.5
- A Model of a Distributed System 33

Figure 4.6
- A Model of a Single Delay due to Locking a File. 35

Figure 4.7
- A Model of a File Disk with Three Transactions Accessing Three Different Files 35

Figure 4.8
- A Model of a Delay for the Sharing of Two Files by Two Transaction Types. 37

Figure 4.9
- Actual Implementation of a Model of a Delay for the Sharing of Two Files by Two Transaction Types 37

List of Figures (cont.)

Figure A.1
- Uniform Distribution. 75

Figure A.2
- Exponential Distribution. 75

List of Tables

Table 3.1
- Simulation Results for Figure 3.1 from AT&T 3B2. . 19

Table 3.2
- Simulation Results for Figure 3.1 from AT&T 3B15 . 21

Table 3.3
- Simulation Results for Figure 3.3. 24

Table 4.1
- Probabilities and Service Times For a One-Host System 40

Table 4.2
- Probabilities For a Two-Host System. 41

Table 4.3
- Probabilities of Transaction Classes Executing on Hosts 45

Table 6.1
- Mean Values of Variates Generated for Various Statistical Distributions. 64

1. Purpose of Paper

This report is an attempt to combine derivations and results from two different studies. The first area is that of simulating a distributed file system using a queueing-network model. The second is that of studying the throughput of various types of networks with various characteristics such as load averages, number of users, and propagation delays, and various probabilities of a host transmitting packets and having packets to transmit. The intent of the report is to use a particular model of a distributed file system, and to "insert" various network delays at the network link in the model. These delays will be calculated through the use of published distributions corresponding to particular types of networks. A distributed version of a Concurrent C simulator will be used to run the different models with varying parameters. The primary purpose of this work is to verify the distributed Concurrent C simulator by comparing results it gives to results obtained in [HAC 1986] using the same models. A secondary purpose of this work will be to derive performance measurements for various types of networks under this queueing network model of a distributed file system.

The model of the distributed file system to be used was introduced in [Hac 1986]. The author uses a queueing-network to model a number of host computers connected by a high-speed network. Each host computer is modeled by server-

queue combinations, one representing the CPU and one representing each file disk on that host. Two other servers are used to model both terminal input and display output. These components are connected together by directed arcs to form what is called a network model. Similar graphs, for representing computer components can be found in [Sauer and Chandy 1981; Sauer and MacNair 1983; Lavenberg and Sauer 1983]. This model is then extended to account for multiple hosts on a carrier-network, where the model of a host computer is the same and the model of the carrier-network is represented by another server-queue combination denoting the delay encountered in transmitting data over the carrier. These two graphs serve as the foundation for the overall model of the distributed file system. But the model is not yet complete. The file system has yet to be incorporated. Files are assumed to already be distributed and the author of [Hac 1986] has decided to model only the delay encountered in accessing a file. This implies placing either a shared or exclusive lock on the file while processing of the file occurs. This delay of locking a file is also modeled in queueing-network form. Once all of the system's components are connected into a queueing-network model, different probabilities are assigned to each of the arcs for the purpose of showing the different paths in the graph that a job might take. These probabilities are derived and calculated by formulas.

The high-speed network in the distributed file system

model is assumed to be a generic network. It is implemented by one server with an associated delay time. This delay time was equated to 0.2 milliseconds in Hac's calculations. This value was derived from a small business installation that served as a source for simulation input data to verify the model. It is at this particular server in the distributed file system model in which various network delays (calculated beforehand for a specific network) will be inserted.

The formulas given in [Takagi and Kleinrock 1985; Kleinrock and Tobagi 1975; Tanenbaum 1981] are used for calculating network capacity. Network throughput can be calculated from network capacity as will be explained later. In modeling delays for processing a file, a fixed file size is assumed. This will allow for the calculation of a delay time to be "inserted" into the carrier-network link of the distributed file system model. Without this assumption there is no way to equate throughput in terms of units of time only.

The formulas in [Takagi and Kleinrock 1985; Kleinrock and Tobagi 1975; Tanenbaum 1981] describe capacity for the following types of networks:

Infinite Users:

(1) Pure ALOHA
(2) Slotted ALOHA
(3) Nonpersistent CSMA
(4) Slotted Nonpersistent CSMA

(5) 1-Persistent CSMA
(6) Slotted 1-Persistent CSMA
(7) Slotted P-Persistent CSMA
(8) 1-Persistent CSMA/CD

Finite Users:
----

(9) Slotted 1-Persistent CSMA
(10) Slotted P-Persistent CSMA

Various parameters are required to calculate the throughput of a network. Some of these inputs include: (1) propagation delay, (2) baud rate (speed) of the network, (3) packet size, (4) number of packets to send, (5) number of users (if applicable), (6) probability of transmission by a user at a station, and (7) probability of a packet arriving to be transmitted. A few of the formulas deal with infinite summations that were solved using approximation techniques. Throughput values are derived for each of the above networks in a simulation of the distributed file system. These throughput values are converted into a delay time to be associated with the network link of the model.

Chapter 2 contains a discussion of the basic concepts of simulation and describes the environment in which this study is taking place. Verification of the simulator is addressed in Chapter 3. The models of a host computer and file access delay are described in Chapter 4, along with the various formulas that determine service times for servers within those models. Chapter 5 discusses the characteristics of various networks and details how both throughput and delay times are calculated. The results of different simu-

lation runs are shown in Chapter 6. Chapter 7 discusses conclusions and future research.

2. Introduction to Simulation

2.1. Simulation Concepts

Before achieving a thorough understanding of this project, one must understand some of the primitive concepts of simulation. This includes relevant terminology, icons used to represent different nodes in a queueing network model (simulation graph), the environment in which this study is taking place, and the simulation process itself. Each of these elements is now described in more detail.

In simulating the flow of "jobs" through any system (i.e. a computer system), one must first identify the elements of the system and how these elements interact. The identified elements will become the "nodes" in the simulation and the interactions will become the "arcs" connecting two nodes. Arcs and nodes are the basic building blocks of a queueing network model. Users must then determine what constitutes a "job" in the system. The identification of jobs, elements of the system, and interaction among these elements will determine the level at which the system is modeled.

Once this modeling level is established, servers (representing service elements of the system being simulated) can be assigned. Queues are usually implemented in front of a server to hold jobs until a server becomes available. From this, servers and queues are usually used

together. A source represents the introduction of jobs into the system, and a sink represents the departure of jobs out of the system. These four nodes, sources, servers, queues, and sinks, are the most commonly used symbols in a queueing network model. These are shown in Figure 2.1. Other nodes capable of representing more complex simulations are shown in Figure 2.2.

A fission node allows one job in the system to become two jobs. A fusion node represents two jobs combining into one job. If a fission node is used in a simulation graph, a fusion node must be used also. If a user desires that a job split into two nodes, and those two jobs will never merge, then a split node is used. (Note: The latter three nodes were not implemented in the simulator used for this project at the time of this writing.)

The path a job takes in a simulation graph can be deterministic (i.e. one path) or probabilistic (i.e. more than one path). A branch node is shown in Figure 2.2. A certain probability is assigned to each of the output paths of the branch node. This probability represents the chance (%) that a job will travel down this path. During an actual simulation run, a random number between 0 and 1 is generated when a job has a choice of paths. The generated number determines which path the job takes. A "path combine" represents the fact that two jobs coming from two different input paths will take the same output path. No probability

SOURCE

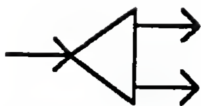
SINK

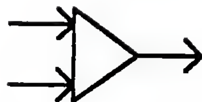
SERVER

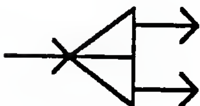
QUEUE

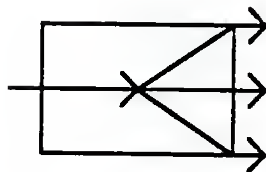
ARC

Figure 2.1 – Basic Nodes in a Simulation Graph









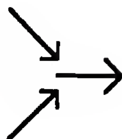


Figure 2.2 - Nodes of More Complex Simulation Graphs

is assigned here. A "path combine" was not implemented at the time of this writing, though its semantics can be implemented by directing all incoming arcs directly into the next node of the simulation graph.

As an example of the various nodes discussed thus far, Figure 2.3 shows a simulation graph taken from [Lavenberg and Sauer 1983] that is representative of four stations organized in a ring. Each station has two server-queue combinations to represent the transmission delay of a message sent in one of the two directions around the network. Probabilities are assigned to each path (arc). S1, S2, S3, and S4 represent the four servers, while SK represents a sink.

In addition to nodes, arcs, and probabilities, another vital element in a simulation includes service/arrival times. An average service time is assigned to each server, and an average interarrival times is assigned to each source. These metrics are derived from real-life systems. Interarrival times reflect how much time passes before a new job is generated (enters the system), while service times reflect the amount of time required to service a job at a particular server. A user of a simulator should know enough about the system being simulated to provide these statistics. If these statistics are not readily available, one must observe the real-life system and then record these figures. Various service and interarrival times would then be added to the derived simulation graph.

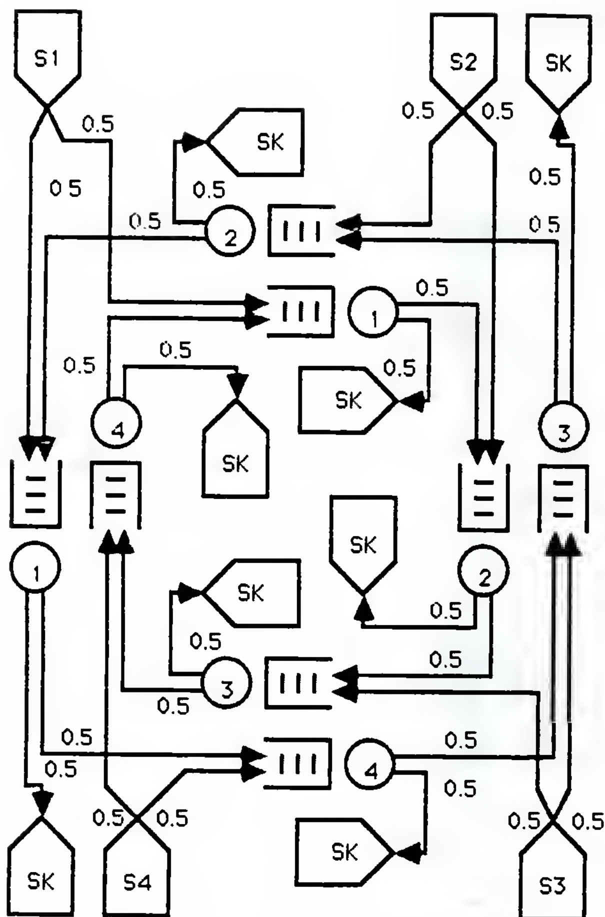


Figure 2.3 - Communications Ring Network

In many instances a fixed service time or interarrival time is not truly representative of the system being modeled; but a user might have an approximation of the average service or interarrival time. In these cases it is desirable to first collect data from the real system, and second to plot this data on a graph. This graph is then compared to various statistical distributions to determine which distribution most closely represents the plotted data. Once a distribution is found, statistical formulas can be used to generate service times or interarrival times, that would represent the graph of the points plotted earlier from the real system. This allows service times or interarrival times to be variable, not fixed. The variation in values is determined by the statistical distribution used. Certain distributions are known to best represent service times or interarrival times for specific types of systems. For a further explanation, refer to [Russell 1983]. A discussion of each statistical distribution implemented in this project is given in Appendix A. The source code implemented for these distributions is shown in Appendix B.

2.2. Simulation Environment

The environment in which this project was conducted included the use of a simulator written in Concurrent C, a deadlock detection algorithm also written in Concurrent C residing underneath the simulator, and a graphical data-

input language supported by LISP. A simulation is carried out by first visualizing the system on the graphical front-end in terms of the icons shown in Figure 2.1. This is achieved through the use of a menu system which allows users to select the desired nodes, move them to a satisfactory location (on the screen), and connect them with arcs. Various probabilities, service times, and interarrival times are then selected, including any probability distributions, if desired. Once all of this data is entered, the graphical front-end translates it into a record-format and sends it to the Concurrent C portion of the simulator via a network. The Distributed Concurrent C simulator interprets it and builds an executable simulation. The simulation is then initiated by the user.

Each node in the simulation graph becomes a Concurrent C process. These processes are distributed over a network to various minicomputers by the use of unique machine numbers. The user specifies the machine on which each process will run. The processes then communicate with each other by using transaction calls built into the Concurrent C language. A simulation clock is kept within each service and arrival node, which represents the current time of the simulation as this particular node sees it. The simulation algorithm checks each node in the graph and finds the one with the lowest simulated time. The service time associated with this node is then added to the old clock value. (As mentioned earlier this service time can be a fixed value or

be generated by a statistical distribution.) The new clock value replaces the old value at this node, and the job is forwarded on to the next node. Jobs are processed throughout the system in this manner until (1) the simulation stop-time, as entered by the user, is reached, or (2) the user stops the simulation through the use of a menu selection.

Due to the simulator's use of Concurrent C processes, deadlock can occur. If the deadlock detection algorithm [Maekawa, Oldehoeft, and Oldehoeft 1987] discovers deadlock, it notifies the simulator which takes measures to correct the situation. Deadlock usually occurs when a particular node in the simulation graph has both an incoming arc and an incoming feedback arc. If a job is not present on both arcs at the time the simulation algorithm is ready to execute a job at this node, deadlock occurs. Recovery is achieved by sending a "null" job into the node from the "empty" arc. The "null" job has no affect on the processing of the "real" job. Processing then continues as usual after the deadlock is broken.

Upon completion of the simulation, various statistical information can be obtained and displayed to the user. These statistics include such metrics as average queue length, number of jobs through a server, average server utilization, etc. These statistics can be obtained in two ways. If the user specifies that the simulation is to stop

at a particular simulated time, then the statistics of the system can be shown. The statistics are also shown when the simulated time reaches the simulation stop time entered by the user.

For a more-detailed explanation of the simulator, deadlock detection algorithm, and graphical input language, see reports to be written by [Vopata 1988], [Hammond 1988], and [Butler 1989] respectively.

3. Verification of the Distributed Concurrent C Simulator

Before assessing the results of any simulation, the simulator must first be verified to indicate that it can indeed simulate reality. To avoid the testing of numerous simulated systems and comparing them to their real counterparts, a more practical approach is taken. Various established simulators are known to exist that can accurately model reality. By comparing output obtained from these established simulators with output obtained from a new simulator, one can determine whether the new simulator closely models reality. The purpose of this approach is not to prove the simulator can model any system perfectly; a more rigorous approach would have to be used. Rather it is an attempt to give an estimation of this simulator's validity and worthiness.

It was decided that three rather simple test cases would be used for comparison purposes. These test cases, taken from [Bulgren 1982], were run in SIMSCRIPT II.5, an established simulation language. Output from each case, such as number of jobs processed, utilization of a server, average arrival and service times, etc. is given. The simulation graphs and various input parameters are shown in Figures 3.1, 3.2, and 3.3.

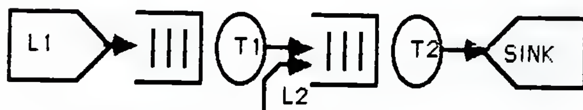
3.1. Simulation Graphs

Arrival Rate (L) = 0.2 minutes
Service Rate (T) = 0.1 minutes

Figure 3.1 - Single-Queue Single-Server System

Arrival Rate (L) = 0.2 minutes
Service Rate (T) = 0.1 minutes

Figure 3.2 - Single-Queue Single-Server Feedback System



Arrival Rate (L1) = 0.4 minutes
Arrival Rate (L2) = 0.4 minutes
Service Rate (T1) = 0.2 minutes
Service Rate (T2) = 0.1 minutes

Figure 3.3 - Traffic Merging System

3.1.1. Single-Server Queue

The first simulation is a single-queue single-server system shown in Figure 3.1. In reality this can model any type of system that provides a single service in which a single waiting-line (queue) forms for that service. The arrival rate into the system is described by the use of an exponential statistical function. The service rate provided by the server is also implemented as an exponential distribution. This function requires one parameter, the average amount of time before an event occurs, i.e. an arrival or a service. The function will then generate values "around" the mean value given as the parameter in an exponential fashion. This function gives a "randomness" to the amount of time that passes before an arrival occurs, or a service is completed. All statistical functions give this "randomness" in the values produced, each in different ways depending upon the function used and the values given as input parameters. The input parameter for arrivals in this example is 0.2, and for service times is 0.1. The results from the SIMSCRIPT simulation run are shown in Tables 3.1 and 3.2.

3.1.2. Single-Server Queue with Feedback

The second simulation is a single-queue single-server feedback system shown in Figure 3.2. In reality this can

Table 3.1

Simulation Results for Figure 3.1
from AT&T 3B2

A. Mean Arrival Time
B. Mean Service Time
C. Utilization of Server
D. Number of Jobs Completed

After 300 minutes:

	SIMSCRIPT	C.C.
A.	0.207	0.217
B.	0.100	0.097
C	0 485	0.449
D	1451	1382

After 600 minutes:

	SIMSCRIPT	C.C.
A.	0.204	0.207
B.	0.099	0.099
C.	0.487	0.476
D.	2943	2900

After 900 minutes:

	SIMSCRIPT	C.C.
A.	0.205	0.201
B.	0.100	0.099
C.	0.486	0.490
D.	4379	4474

After 1200 minutes:

	SIMSCRIPT	C.C.
A.	0.204	0.201
B.	0.099	0.099
C.	0.484	0.495
D.	5892	5970

After 1500 minutes:

	SIMSCRIPT	C.C.
A.	0.204	0.202
B.	0.099	0.100
C.	0.488	0.496
D.	7361	7428

After 1800 minutes:

	SIMSCRIPT	C.C.
A.	0.202	0.202
B.	0.100	0.100
C.	0.493	0.496
D.	8905	8915

Table 3.1 (cont.)

Simulation Results for Figure 3.1
from AT&T 3B2

A. Mean Arrival Time
B. Mean Service Time
C. Utilization of Server
D. Number of Jobs Completed

After 2100 minutes:

	SIMSCRIPT	C.C.
A.	0.202	0.201
B.	0.100	0.099
C.	0.495	0.494
D.	10413	10457

After 2400 minutes:

	SIMSCRIPT	C.C.
A.	0.201	0.201
B.	0.100	0.100
C.	0.499	0.494
D.	11950	11920

After 2700 minutes:

	SIMSCRIPT	C.C.
A.	0.201	0.201
B.	0.100	0.100
C.	0.499	0.495
D.	13458	13402

After 3000 minutes:

	SIMSCRIPT	C.C.
A.	0.200	0.200
B.	0.100	0.100
C.	0.501	0.499
D.	14982	14994

Table 3.2

Simulation Results for Figure 3.1
from AT&T 3B15

A. Mean Arrival Time
B. Mean Service Time
C. Utilization of Server
D. Number of Jobs Completed

After 300 minutes:

	SIMSCRIPT	C.C.
A	0.207	0.199
B.	0.100	0.102
C.	0.485	0.512
D	1451	1510

After 600 minutes:

	SIMSCRIPT	C.C.
A.	0.204	0.196
B.	0.099	0.101
C.	0.487	0.516
D.	2943	3054

After 900 minutes:

	SIMSCRIPT	C.C.
A.	0.205	0.198
B.	0.100	0.103
C.	0.486	0.519
D.	4379	4532

After 1200 minutes:

	SIMSCRIPT	C.C.
A.	0.204	0.199
B.	0.099	0.102
C.	0.484	0.513
D.	5892	6033

After 1500 minutes:

	SIMSCRIPT	C.C.
A.	0.204	0.198
B.	0.099	0.102
C.	0.488	0.513
D.	7361	7566

After 1800 minutes:

	SIMSCRIPT	C.C.
A.	0.202	0.198
B.	0.100	0.102
C.	0.493	0.514
D.	8905	9091

Table 3.2 (cont.)

Simulation Results for Figure 3.1
from AT&T 3B15

A. Mean Arrival Time
B. Mean Service Time
C. Utilization of Server
D. Number of Jobs Completed

After 2100 minutes:

	SIMSCRIPT	C.C.
A.	0.202	0.198
B.	0.100	0.102
C.	0.495	0.513
D.	10413	10602

After 2700 minutes:

	SIMSCRIPT	C.C.
A.	0.201	0.199
B.	0.100	0.101
C.	0.499	0.508
D.	13458	13551

After 2400 minutes:

	SIMSCRIPT	C.C.
A.	0.201	0.198
B.	0.100	0.101
C.	0.499	0.512
D.	11950	12101

After 3000 minutes:

	SIMSCRIPT	C.C.
A.	0.200	0.199
B.	0.100	0.101
C.	0.501	0.508
D.	14982	15072

model any type of system similar to the first example in which a "job" in the system might return to the server for more service. A particular probability is assigned to a job "feeding back" into the system. These two probabilities should sum to 1.0. An exponential function describes the arrival rate and service rate of the system. The input parameters to these functions are the same as the first example, and the probability of returning for service is 0.3.

3.1.3. Traffic-Merging System

The third simulation represents a traffic-merging system in which jobs may enter the system from two sources, each at a different arrival rate, if desired. The simulation graph is shown in Figure 3.3. Exponential functions describe arrival rates and service rates in the system. The parameters to the exponential function for both arrival rates are 0.4. The service rate of the first server is 0.2; the service rate of the second server is 0.1. The results of this simulation are shown in Table 3.3.

3.2. Results of Verification

At the time of this writing, many of the features to allow the Concurrent C simulator to distribute its processing over many different computers were not implemented.

Table 3.3

Simulation Results for Figure 3.3

A. Mean Arrival Time
B. Mean Service Time
C. Utilization of Server
D. Number of Jobs Completed

After 9000 jobs were generated by T2:

AT&T 382 Results:

Server/Queue T1

	SIMSCRIPT	C.C.
A.	0.404	0.397
B.	0.204	0.198
C.	0.505	0.499
D.	4534	9078

Server/Queue T2

	SIMSCRIPT	C.C.
A.	0.410	0.400
B.	0 100	0.100
C.	0.492	0.503
D.	9000	9000

AT&T 3815 Results:

Server/Queue T1

	SIMSCRIPT	C.C.
A.	0.404	0.396
B.	0.204	0.198
C.	0.505	0.501
D.	4534	9858

Server/Queue T2

	SIMSCRIPT	C.C.
A.	0.410	0.401
B.	0.100	0.100
C.	0.492	0.485
D.	9000	9000

This, however, did not prevent the simulation algorithm from being run on just one machine as long as all nodes in a simulation graph were set up to be executed on this one machine. So the verification values shown were derived by simulating the graphs in Figures 3.1, 3.2, and 3.3 on one machine only. Values are shown for two different simulation runs of each of these systems, where each run occurred on a different machine.

The results given in Table 3.1 were obtained by running the simulation algorithm on an AT&T 3B2 computer. The results shown in Table 3.2 were obtained after the simulation algorithm was run on an AT&T 3B15 minicomputer. The results show appropriate values at 300-minute intervals for ten intervals.

No results were given from the Concurrent C simulator for the simulation graph shown in Figure 3.2, due to the fact that the only statistic given by [Bulgren 1982] is the average time a job spends in the queue. This particular statistic had not been implemented within the Concurrent C simulator at the time of this writing.

Results for the simulation graph in Figure 3.3 are shown in Table 3.3. These values were derived at the end of a 3000-minute simulation period. It is difficult to truly compare the values given by both simulators in this latter case due to the fact that the values generated by the SIMSCRIPT simulator were given after 9000 jobs were generated

by the second source. The Concurrent C simulator did not have such a feature to halt the simulation after a certain number of jobs were generated at the time of these simulation runs. The statistics generated were produced every 300 minutes. The values used to compare against the SIMSCRIPT simulator were those given at the next 300-minute interval after 9000 jobs were generated by the second source.

From the results that were obtained, it appears that the Concurrent C simulator generates similar values against the SIMSCRIPT simulator. Service times and interarrivals times are relatively the same in most instances. An exponential function was used to describe these two time values. The number of jobs serviced per time period seems to be slightly more in the Concurrent C simulator. Utilization of each server seems to be about equal in both simulators. The average amount of time a job spent in the queue in the SIMSCRIPT simulator could not be compared to the Concurrent C simulator since this feature had not been implemented in the latter simulator.

As mentioned before, a more rigorous approach would have to be taken to prove the simulator is correct. These simple test cases indicate that the Concurrent C simulator can generate values that are comparable to values generated by an established simulator.

4. Distributed File System

4.1. Modeling as a Queueing Network

4.1.1. Developing a Model of a Host

4.1.1.1. A Model of a Simple Computer

In [Hac 1986], a model of a distributed file system with dynamic file-locking is derived through a series of enhancements to a model of a simple computer. The simulation graph of Figure 4.1 shows three of the main components in a computer. Jobs within this system are generated at the terminals and flow towards the CPU. The CPU accepts the job's request for work and calls upon the file disk to make the appropriate file available. Once the file is available, the CPU will process the request and the job may (1) return to the terminals for more input and thus more processing at the CPU, (2) return to the terminals as a sign of completion of the job, or (3) advance to the file disk again in need of another file. This model can handle many different transactions as long as each transaction does not access the same file as another transaction. In other words this model does not support the concept of locking a file that would cause other transactions needing the locked file to encounter some delay. This "locking" feature more accurately represents the actions of a computer system, and is implemented in upgraded versions of this computer simulation graph.

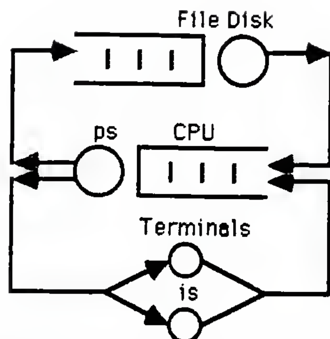


Figure 4.1 - Simple Model of a System

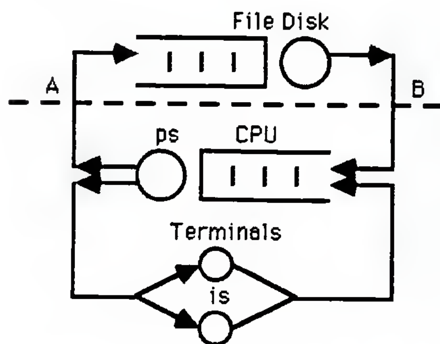


Figure 4.2 - Simple Model of a System
with Response Time Line

4.1.1.2. Enhanced Models

In Figure 4.2, a dotted line has been added to the previous figure. If one transaction should lock a file at the disk and then travel to other service centers, then the amount of time this transaction spends within the system is shown below line AB. In other words if a transaction begins file execution after acquiring the needed files, completes execution and unlocks the file at the CPU, the difference in these two times is represented in the system by a job trying to reach point A from point B. If this delay time were known, it could be incorporated into the simulation graph as in Figure 4.3 to represent the delay encountered by a second transaction needing the first transaction's locked file. From this a certain probability, p, could be assigned to the path into the delay server. This probability would represent the chances of a transaction encountering a delay in accessing a desired file. The other path, representing no delay, would be assigned a probability equal to (1-p).

There is one problem with this particular model. The delay associated with the delay server would have to be calculated iteratively every time a new job was serviced. This is because the response time of the system, of which this server represents, would change each time a job was serviced as a result of the fluctuating load on all system components. It would be more desirable to have an estimated mean value that would be representative of the delay

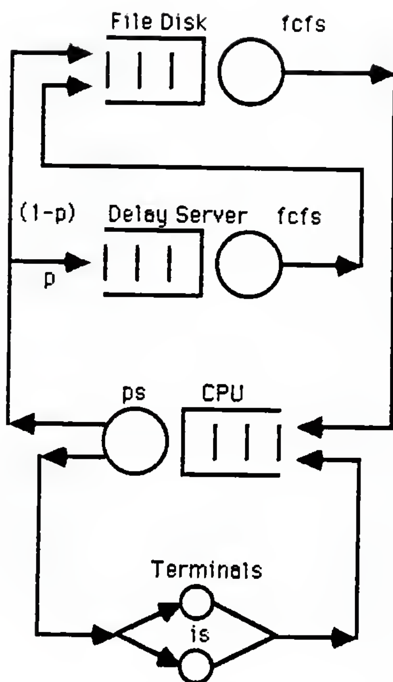


Figure 4.3 - A Model of a Host with a Delay Server

encountered for one specific type of transaction due to the load on all system components. In other words each transaction type would have its own delay time for waiting on transactions of the same type (class). Transaction using the same resources, in this case files, are considered to be in the same class. This delay time calculation would be based on the service loads of all elements in the system. Such a system might be depicted as in Figure 4.4. In this particular simulation graph of a host computer, two file disks are represented which are each accessed by one transaction class. A display server has been added to the two previous figures to represent transactions performing display outputs. This is the model of a host used in [Hac 1986].

Once a model for each host is assembled, as in Figure 4.4, a distributed system of these hosts can be created as shown in Figure 4.5. A network server is inserted between the hosts to represent the delay encountered in transmitting data (files) over a network. It is at this point in the overall model of the distributed file system in which different service times will be inserted to represent various network protocols. The discussion of networks and assumptions therein that lead to the derivation of this service time are set forth later in Chapter 5.

4.1.2. Developing a Model of a Delay

4.1.2.1. No File-Sharing

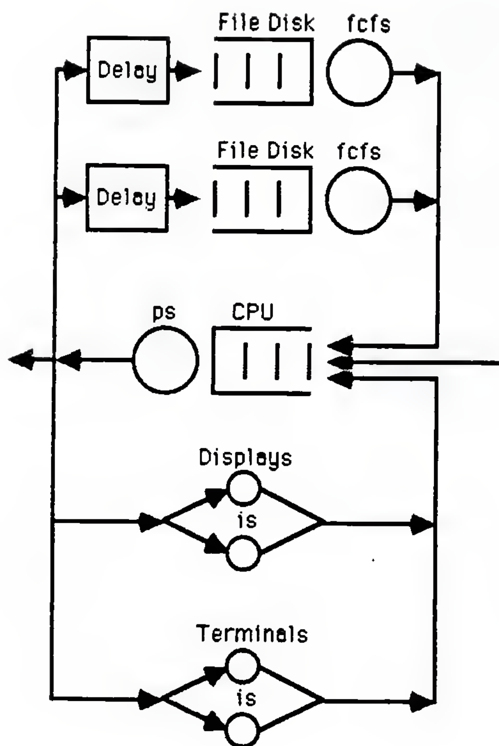


Figure 4.4 - A Model of a Host

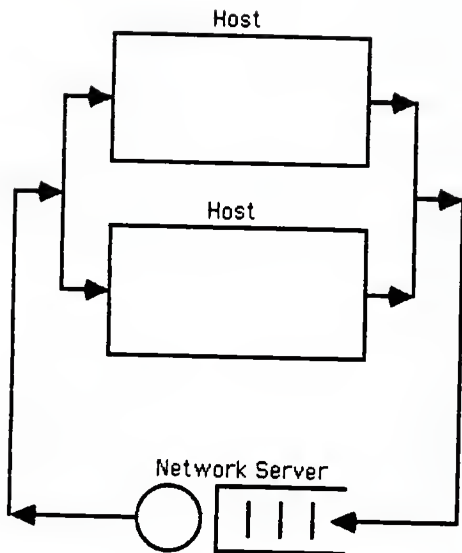


Figure 4.5 - A Model of a Distributed System

[Hac 1986] also derives a model that represents the delay encountered in accessing a file. Figure 4.6 shows the breakdown of a single delay due to locking a file. This simulation graph is inserted in the delay node in Figure 4.4. The various service centers in the model of a delay will have service times associated with them representative of the load on all system components in the overall simulation graph. The CPU delay represents the delay one transaction experiences upon finding a needed file locked, and the transaction which locked the file is executing in the CPU. According to [Hac 1986], the mean service time associated with this server represents an estimation of the remaining time that the transaction that locked the file spends in the CPU. The display delay and terminal delay are defined similarly as the delay one transaction experiences as the result of another transaction (which locked the file) executing in the display and terminal respectively. In other words these nodes are representing the time required for the transaction which locked the file to terminate so the delayed transaction can execute. The model of a delay can be used many times in an overall simulation graph as in Figure 4.7. This graph represents three separate transactions accessing three separate files where no file is shared among the transactions.

4.1.2.2. File-Sharing

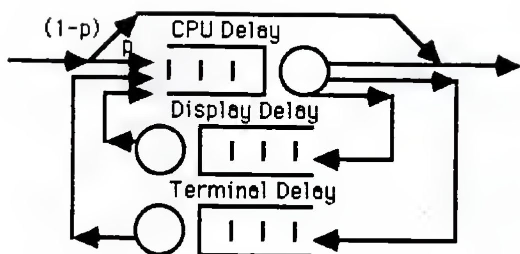


Figure 4.6 - A Model of a Single Delay due to Locking a File

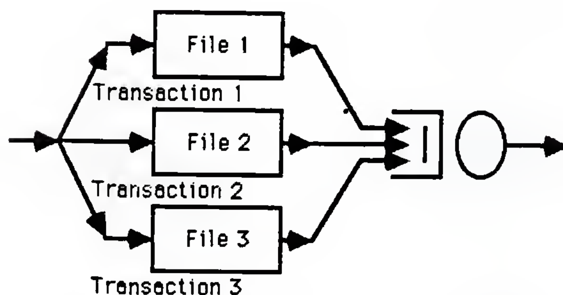


Figure 4.7 - A Model of a File Disk with Three Transactions Accessing Three Different Files

If a file is shared among transactions, a simulation graph similar to Figure 4.8 would be used. This graph represents the fact that the second file is shared by the first two transaction classes. The two extra delays represent competition for the desired file against transactions of the same class, and against transactions of the other class. It would appear that the first delay encountered by each transaction for the second file already represents the delay experienced due to competition among transactions of the same class. If this model was implemented, the job type would have to be distinguished to determine which of the two extra delays should be bypassed. At the time of this writing the simulator has no features for distinguishing job types. So another equivalent model was used. The model for the sharing of two files that was actually implemented appears in Figure 4.9.

4.1.3. Formulas for Service Times of Delay and Host Components

As mentioned earlier, the service times associated with each server in the overall simulation graph are based upon the load on all system components. These service times can be calculated by formulas given in [Hac 1986]. The formulas describe the service times of a display and a terminal in the model of a host, and service times of a display, a terminal, and the CPU in the model of a delay. Though no

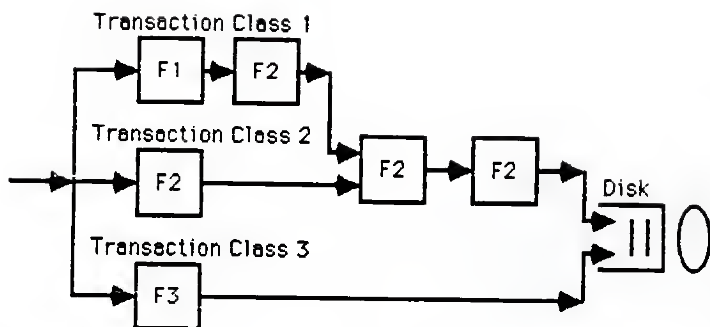


Figure 4.8 - A Model of a Delay for the Sharing of Two Files by Two Transaction Types

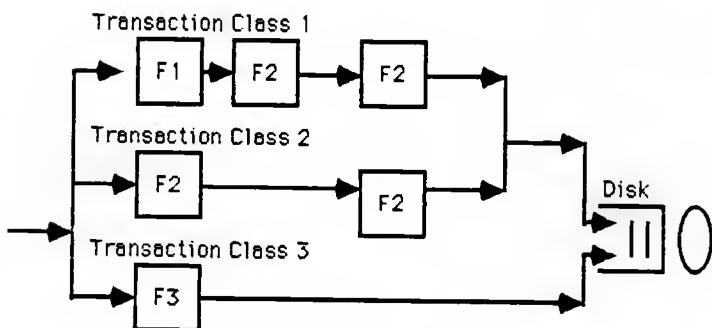


Figure 4.9 - Actual Implementation of a Model of a Delay for the Sharing of Two Files by Two Transaction Types

derivation of these formulas was given in [Hac 1986], a brief explanation is now given.

The service times for the delay components are dependent upon a number of factors within the system. Each transaction in the system belongs to a specific transaction class. Each class has data associated with it representing the amount of time required to service a transaction of this type at each service center, i.e. display, terminal, and CPU. It is important to note the distinction between one transaction and one transaction class. The number of transactions from each class and the number of transaction classes help determine the service time for each individual service center. Probabilities of these transaction classes traveling to specific service centers also play a role in the service time calculation. From all of the factors described above comes a service time for each service center that is representative of the load on ALL service centers.

The terms and formulas for the components in the model of a delay are shown in Appendix C. The terms and formulas for terminals and displays in the model of a host are shown in Appendix D. The actual implementation of the formulas described in Appendices C and D is given in Appendix E.

4.2. Implementation

The individual service times and probabilities for each

transaction class used in [Hac 1986] are shown in Tables 4.1 and 4.2 for a one-host and a two-host system respectively. Appropriate values were taken from these tables and entered as input into the program shown in Appendix E for calculating average service times and probabilities representative of the entire system load.

The input format for a one-host system with no file-sharing is shown in Appendix F. The input format for a one-host system with file-sharing, and for a two-host system is similar. Appendix G gives the input format of a two-host system with file-sharing. The only value not given in Tables 4.1 or 4.2 that is needed for input into the service times program (shown in Appendix E) is the number of "active" transactions for each transaction class. An active transaction is defined as a job that is executing on a particular host. With the possibility of accessing remote hosts, the number of active transactions on an individual host should vary over time. In these simulation runs, it is assumed that each transaction class has a number of active transactions equal to the number of transactions within that class.

The output from the service times program echoes all user input before giving its results. Since many data values are entered, users should verify these input figures before accepting the results as correct. This output includes service times for both the terminal and display

Estimated Mean Values of the Model Parameters For
A Single-Host System

Transaction Type (Class)	CPU Time [ms]	"Think" Time [ms]	Display Time [ms]
1	42	1000	15
2	35	10000	25
3	224	1000	20
4	32	500	15
5	15	2000	15

Transaction Type (Class)	Probability of Disk Access	Probability of Terminal Access	Probability of Display Access
1	0.4	0.11	0.49
2	0.8	0.01	0.19
3	0.75	0.05	0.20
4	0.966	0.007	0.027
5	0.3	0.09	0.61

Table 4.1

Estimated Mean Values of the Model Parameters For
A Two-Host System

Transaction Type (Class)	Probabilities			
	Local Disk Access	Local Display Access	Local Terminal Access	Remote Host Access
1	0.39	0.47	0.10	0.04
2	0.74	0.17	0.02	0.07
3	0.70	0.19	0.04	0.07
4	0.88	0.025	0.007	0.088
5	0.29	0.59	0.09	0.03

Transaction Type (Class)	Probability of Remote Host Disk Access	Probability of Exiting The Remote Host
1	0.79	0.21
2	0.987	0.013
3	0.943	0.057
4	0.993	0.007
5	0.778	0.222

Table 4.2

server in the model of a host, and the service time for each component in the model of a delay for each transaction type. Probabilities are also given for each transaction type experiencing a delay in accessing a file. Other information gives the probability of each transaction class accessing the CPU after completing service at the file disk. Although these probabilities are not assigned to any branches on the simulation graph, they provide statistics on the chances of the next job traveling to the CPU from the file disk being of a certain transaction class. Probabilities of returning to a particular host are also given; but with only two hosts it is obvious that these values will always be one.

4.2.1. One Host

In modeling a one-host system, many variations were implemented as was done in [Hac 1986]. Three transaction classes were assumed. The probability of encountering a delay at a file disk was set to three different values of 0.1, 0.25, and 0.5. Files on the file disk were assumed not shared in the first three cases and shared in the last three cases. The simulation graph in the no-file-sharing case resembled the model of a one-host system, shown in Figure 4.4, coupled with the model of a delay, shown in Figure 4.6. The simulation graph of the file-sharing case replaces Figure 4.6 with Figure 4.7. The values given from the service times program were then added to each simulation graph

appropriately.

4.2.2. Two Hosts with a Generic Network

In modeling a two-host system, the generic network value of 0.2 milliseconds used in [Hac 1986] was implemented first to determine how well the statistics from the original simulation agreed with those given by the Distributed Concurrent C simulator. Three transactions classes were assumed on each host. The values for the fifth transaction class shown in Table 4.2 were used as the third transaction class on each host. Files were assumed to be shared in three cases and not shared in three cases, as before, with a probability of delay of 0.1, 0.25, and 0.5. The simulation graphs used were two hosts connected by a network server as shown in Figure 4.5. One host had a file shared between two transactions in the file-sharing cases. Again values given from the service times program were then added to each simulation graph appropriately.

In [Hac 1986] the formulas representing remote transactions accessing and leaving a local host are assumed to be valid only if the remote transactions are statistically identical in the local host. These "identical statistics" refer to service times and probabilities associated with the model of a delay for a remote transaction. So the service times assigned each element in the delay model in the transaction's local host would also be assigned to each

element in the delay model in the transaction's remote host. Since remote transactions do not travel to displays or terminals in the remote host, the only probabilities assigned to branch nodes for these transactions in the remote host are for traveling to the remote file disk or back across the network to another host.

There are a few values that are not generated by the service times program that are needed on the simulation graph. These are the values for the percentage of local and remote jobs executing on each host. After a job is generated from the source, it basically loses its identity. At present the simulator has no feature to attach an attribute onto the job to identify which transaction class it belongs to, etc. So after each job exits the CPU, a random number is generated that determines if the job is local or remote. The distribution for this generation is calculated in Table 4.3. Host 1 has a 94% probability that the job exiting the CPU is a local transaction. On Host 2 the probability for a local transaction exiting the CPU is almost 96%. These probabilities would be assigned to the first branch node entered after the CPU. After the locale of the job has been calculated, another branch node is implemented that determines to which transaction class the job belongs. Since there are three classes of local transactions, each transaction class was assigned a 1/3 probability of occurrence. Each of the three remote transaction classes was also assigned a 1/3 probability of occurrence. In other words,

Probability of Each Transaction Class
Executing on Each Host

Class	Host 1 Probability	Host 2 Probability
1	1.000	0.040
2	1.000	0.070
3	0.070	1.000
4	0.088	1.000
5	1.000	0.030
6	0.030	1.000
	3.188	3.140

Host 1:

Local Transactions: 3.000 / 3.188 = 0.941029
Remote Transactions: 0.188 / 3.188 = 0.058971

Host 2:

Local Transactions: 3.000 / 3.140 = 0.955414
Remote Transactions: 0.140 / 3.140 = 0.044586

Table 4.3

the job just processed at the first branch node has an equal chance of belonging to any one of the three local or three remote transaction classes. The probability breakdown at the third level of branch nodes would be the normal breakdown of a job traveling to the other service nodes. Again, these probabilities are shown in Table 4.2.

4.2.3. Two Hosts with Various Networks

After a two-host system was compared to the calculations in [Hac 1986], other values for the service time of the network server were inserted. These service times represented the delay encountered through the use of different protocols sending different numbers of packets over a network. The probability of delay was varied also for each network to determine how this affected system performance. File-sharing and no file-sharing cases were tested.

4.2.4. Input to the Simulator Program

Once each simulation case was organized graphically and all of the associated probabilities and services times were inserted, this information was ready to be used as input into the simulator. At the time of the simulation runs, some of the features of the graphics front-end were still in development which did not allow the data to be inputed in a graphical form. To run the simulations would require the

conversion of the data into a record format that the simulator program could understand directly. This format is the same format into which the graphics front-end would translate its graphical input for the simulator program. An example of the input submitted to the Distributed Concurrent C Simulator along with an explanation of each parameter is shown in Appendix H. The results of all of the simulation runs are given in Chapter 6, after a description is given in Chapter 5 of the network protocols that were simulated in the test cases of two hosts.

5. Networks

5.1. Characteristics of Networks

The intent of this report was to derive some performance measurements on how this model of a distributed file system would be affected if various types of networks were used. Now that the distributed file system model has been explained, a description of each type of network implemented, shown below, will be given.

Infinite Users:

(1) Pure ALOHA
(2) Slotted ALOHA
(3) Nonpersistent CSMA
(4) Slotted Nonpersistent CSMA
(5) 1-Persistent CSMA
(6) Slotted 1-Persistent CSMA
(7) Slotted P-Persistent CSMA
(8) 1-Persistent CSMA/CD

Finite Users:

(9) Slotted 1-Persistent CSMA
(10) Slotted P-Persistent CSMA

5.1.1. ALOHA

The first two types of networks above are ALOHA networks. These network protocols (i.e. rules of communication between two host computers) were developed at the University of Hawaii where they were used in radio packet broadcasting [Tanenbaum 1981]. Pure ALOHA allows users to transmit small

groups of data, known as packets, at any time over a shared channel, in this case, a radio frequency. In later developments a satellite channel was used. If two stations begin transmission at roughly the same time, a collision will occur and the data transmitted by both stations will be garbled. To correct this, after transmitting a group of packets, a station would wait an amount of time equal to two times the propagation delay of the channel, about 540 milliseconds. If no acknowledgement message was received within 540 milliseconds, the transmitting station would assume that a collision had occurred, wait a random amount of time, and then send the packets again. This protocol led to many other protocols which use the same principles and improve on the network's performance. Pure ALOHA is named so because users can begin their initial transmissions at any time.

In contrast, Slotted ALOHA requires users to only transmit at certain points in time. The amount of time between any two consecutive points is known as a slot. The time differential that makes a slot is usually set equal to the time required to transmit a packet. From this, if two packets collide, then they have both filled the same time slot. Thus less bandwidth is used if a collision occurs, and the efficiency of the channel is increased. In other words the amount of time taken up by a collision decreases. Since more packets can be placed on the channel (in this extra amount of time) the channel's efficiency is increased.

It can be shown [Kleinrock and Tobagi 1975] that the efficiency doubles in Slotted ALOHA over Pure ALOHA.

The formulas for calculating throughput, S, for both of these protocols are shown below under the assumption that the interarrival time of the packets to be transmitted follows a Poisson distribution. Statistically this means that the arrival of one packet is independent of the arrival of all other packets.

(1) Pure ALOHA: $S = G * \exp(-2 * G)$

(2) Slotted ALOHA: $S = G * \exp(-G)$

The maximum throughput for (1) occurs at an offered traffic rate, G, equal to 0.5, where S equals 18.4% efficiency. For (2), G is equal to 1.0, and the maximum throughput is 36.8%. The offered traffic rate is defined in [Kleinrock and Tobagi 1975] to be equal to the mean number of packets offered to the channel to be transmitted. This includes both newly-generated packets, and collided packets requiring retransmission. G is equal to the number of packets available per transmission time. As the load on the network increases, G increases accordingly.

5.1.2. CSMA

5.1.2.1. Nonpersistent CSMA

A CSMA (Carrier-Sense Multiple Access) protocol is

somewhat different than the ALOHA protocol and is used between local ground-to-ground stations. CSMA has the capability of determining whether the channel is currently being used; that is it has the ability to sense if the channel is idle or busy. If the channel is idle, then a station can begin transmission; otherwise it reschedules transmission at a later random time. If a collision occurs between two stations which reside a distance apart farther than one packet-transmission time, each station will not receive their respective acknowledgement messages. In other words if two stations begin transmission at roughly the same time, and the time required to send a packet between these two stations is greater than the time required to transmit a packet, then a collision will occur. This is due to the fact that neither station senses that the other has begun a transmission until AFTER it has transmitted one packet itself. Therefore both stations have a packet on the channel at the same time and a collision occurs. Each station will then wait a random amount of time and retransmit. This implementation of CSMA is known as Nonpersistent CSMA. It is nonpersistent because a station does not persist in transmitting its packets if the channel is sensed to be busy. Other variations of CSMA are persistent and are discussed in more detail later.

The throughput, S, of a Nonpersistent CSMA network is given by the following formula:

$$X = G * \exp(-a * G)$$
$$Y = (G * [1 + (2 * a)]) + (\exp[-a * G])$$
$$S = X / Y$$

where G is the offered traffic rate and 'a' is the propagation delay of the network. The maximum throughput varies higher as the propagation delay decreases. At a = 0.01 and G = 10.0, S maximizes at 81.5% efficiency.

Just as in the ALOHA protocols, a pure and slotted version was derived for CSMA. In a Slotted Nonpersistent CSMA network, stations begin their transmissions only on the beginning of a slot time. The slot time is again equal to the packet transmission time. If two packets collide, then they will overlap entirely. Recovery from a collision includes waiting a random amount of time and retransmitting. Efficiency of the channel is increased slightly due to the exact overlap of collided packets and the decrease in bandwidth use thereof.

The formula for throughput, S, of a Slotted Nonpersistent CSMA network is given below:

$$X = a * G * \exp(-a * G)$$
$$Y = a + (1 - \exp[-a * G])$$
$$S = X / Y$$

Again as the propagation delay decreases, the efficiency of the channel increases. At a = 0.01 and G = 10.0, the throughput maximizes with S = 85.7% efficiency. Both formulas for nonpersistent CSMA protocols assume an interarrival

time between packets that is described by a Poisson statistical distribution.

5.1.2.2. Persistent CSMA

Persistent CSMA protocols are subdivided into two categories. The "level of persistence" of these protocols is described by a probability, p. If p = 1, a special case arises and these protocols are known as 1-persistent. Otherwise the protocols are known as p-persistent. The persistence level describes how often a station attempts to transmit a packet on the network, first sensed as busy and now sensed as idle again. In other words if a 1-persistent protocol was used, the random delay normally implemented after (1) sensing the channel busy or (2) a collision, is eliminated. In this case, the station would always transmit immediately (persist with probability 1) after sensing the channel had become idle again. If a p-persistent protocol was used, the station would transmit (persist) immediately after sensing the channel idle again only (p * 100)% of the time. It would delay transmission [(1-p) * 100]% of the time.

According to [Kleinrock and Tobagi 1975] the intent of a 1-persistent protocol design was that the channel would never go unused if it was idle and a station was ready to transmit. The consequence of the design is that all stations now sense the channel as being idle at the same time,

and all transmit with probability one. This can cause the number of collisions to increase dramatically. It is for this reason that p-persistent protocols were designed. P-persistent protocols try to maximize the usage of the channel when it is sensed as being idle, and minimize the number of collisions that occur as stations retransmit.

The other two cases of persistent CSMA protocols are similar to the slotted versions of the protocols discussed earlier. Here also, a station begins transmission only at the beginning of a slot time. The characteristics of 1-persistent CSMA and p-persistent CSMA still hold in the slotted protocols. The formula for throughput of a network implemented with a 1-persistent CSMA protocol is described as:

```
Term1 = G * (1 +  G + [(a*G) * (1 + G + [a*G/2])])
Term2 = 1 * exp(-G * [1 + (2*a)])

Numerator = Term1 * Term2

Term3 = G * (1 + [2*a])
Term4 = 1 - (exp[-a * G])
Term5 = (1 + [a*G]) * exp(-G * [1+a])

Denominator = Term3 - Term4 + Term5

S = Numerator / Denominator
```

The value of S maximizes at an offered traffic rate, G, equal to 1.0, and a propagation delay, 'a' is equal to 0.01, where S is equal to 52.9% efficiency. As in all networks, as the propagation delay decreases the efficiency increases.

The formula for throughput of a slotted 1-persistent

CSMA protocol is given as:

```
Term1 = G * exp(-G * [1+a])
Term2 = 1 + a - exp(-a * G)

Numerator = Term1 * Term2

Term3 = (1+a) * (1 - exp[-a * G])
Term4 = a * exp(-G * [1+a])

Denominator = Term3 + Term4

S = Numerator / Denominator
```

The value of S maximizes at an offered traffic rate, G, equal to 1.0, and a propagation delay, 'a', is equal to 0.01, where S is equal to 53.1% efficiency.

The formula for throughput of a slotted p-persistent CSMA protocol is given by the following equations:

```
Term1a = p * ([1 - p] ** k)
Term1b = a * (1 - [(1 - p) ** (k + 1)])
Term1c = G * ([1 - p] ** [k + 1])
Term1d = (1 - [(1 - p) ** (k + 2)]) / p
Term1e = - (k + 1)
Term1f = a * G * (Term1d + Term1e)
Term1  = (Term1a + Term1b) exp(Term1c + Term1f)

Numerator = G * [for (k=0 to infinity)
                 summation of {Term1}]

Term2a = (1 + a) exp([1 + a] * G)
Term2b = G * ([1 - p] ** k)
Term2c = (1 - [(1 - p) ** (k + 1)]) / p
Term2d = Term2b + (a * G * [Term2c - k])
Term2e = a * [for (k=1 to infinity)
             summation of {exp(Term2d}]

Denominator = Term2a + Term2e

S = Numerator / Denominator
```

The value of S maximizes at different values as the value of the persistence level, p, varies. With a propagation delay

of 0.01, S maximizes at 79.1% efficiency with a persistence level of 0.1. With p = 0.03, S maximizes at 82.7% efficiency.

All of the formulas above are valid under the assumption that there are an infinite number of users. The following two formulas are for a finite number of users. The first formula describes a slotted 1-persistent CSMA protocol with 'm' users.

```
Term1a = m * ([1 - g] ** [(m - 1) * (1 + [1/a])])
Term1b = 1 - ([1 - g] ** [1 + (1/a)])
Term1c = 1 - ([1 - g] ** m)
Term1d = g * ([1 - g] ** [m + (1/a)])

Numerator = Term1a * (Term1b * Term1c + Term1d)

Term2a = (1 + a) * (1 - [(1 - g) ** m])
Term2b = a * ([1 - g] ** [m * (1 + [1/a])])

Denominator = Term2a + Term2b

S = Numerator / Denominator
```

The second formula describes a slotted p-persistent CSMA protocol with 'm' users.

```
Term1a = (1 - p) ** k
Term1b = (1 - g) ** (1 + [1/a])
Term1c = p * ([1 - p] ** k)
Term1d = g * ([1 - g] ** k)
Term1e = p - g
Term1  = Term1a - Term1b * [(Term1c - Term1d) / Term1e]

Term2a = (1 - p) ** (k + 1)
Term2b = (1 - g) ** (1 + [1/a])
Term2c = (1 - g) ** (k + 1)
Term2d = (Term2a - Term2c) / (p - g)
Term2e = Term2a - (Term2b * p * Term2d)
Term2  = Term2e ** (m - 1)

Numerator = p * m * (for [k=0 to infinity]
                      summation of {Term1 * Term2})
```

```
Term3a = (1 - p) ** k
Term3b = (1 - g) ** (1 + [1/a])
Term3c = (1 - g) ** k
Term3d = (Term3a - Term3c) / (p - g)
Term3e = Term3a - (Term3b * p * Term3d)
Term3  = for (k=1 to infinity)
              summation of {Term3e ** m}

Denominator = 1 + a + (a * Term3)

S = Numerator / Denominator
```

5.1.3. Collision Detection

Ethernet, described by [Shotwell 1985], is the name of a product marketed by the Xerox Corporation. It uses a CSMA protocol that has a collision detection mechanism built into it. Thus it is denoted as CSMA/CD. In all of the other CSMA protocols, a collision was detected, or assumed to occur, respectively, if a negative acknowledgement was received, or no acknowledgement was received within a certain time-out/retransmit time frame. With collision detection built-in, a station sends its message to a second station, waits a time equal to two times the propagation delay of the channel, upon which it can read the data it transmitted, as it is propagated back to the sending station. If the data is the same as it transmitted, the station assumes that its message indeed did get through. If the data read is not the same as it transmitted, a collision is assumed to have occurred. The station waits until it senses the channel is idle again, and retransmits without delay. In other words, Ethernet uses a 1-persistent CSMA/CD protocol. The

following formula describes a 1-persistent CSMA/CD protocol where 'b' represents the collision detection factor, or the amount of time required to abort transmission after recognizing a collision. According to [Franta and Chlamtac 1981], if the protocol is slotted, 'b' represents the number of slots (or slot times) before all colliding nodes sense and terminate their transmissions. If the protocol is not slotted, 'b' represents the number of propagation-delay-times or "a-slots" (a is the variable used to represent propagation delay) until all colliding nodes sense and terminate their transmissions.

```
Term1a = G * ([1 + G] exp[-G * (1 + [2 * a])])
Term1b = G * exp(-G * [b + a])
Term1c = ([b * G / 2] + 1) * (1 - exp[-2 * a * G])
Term1d = (a * G * exp[-2 * a * G]) / 2

Numerator = Term1a + (Term1b * [Term1c - Term1d])

Term2a = exp(-G * [1 + a])
Term2b = G * exp(-a * G)
Term2c = (1 - exp[-a * G]) * (1 + [b + a] * G)
Term2d = exp(-b * G) * ([1 - exp(-2 * a * G)] / 2)

Denominator = Term2a + Term2b + Term2c + Term2d

S = Numerator / Denominator
```

5.2. Throughput Derivation

After the formulas in [Kleinrock and Tobagi 1975] were implemented, it was discovered that these formulas did not actually give a throughput value in packets per second. Rather, a capacity value (related to throughput) was given. A few definitions taken from [Franta and Chlamtac 1981]

should make the distinction between these two terms clear.

Channel Utilization (U) is defined as the long-run fraction of time the channel is engaged in the transfer of data. Channel capacity (C) is defined as the maximum possible value of U allowed by the operational characteristics of the protocols employed. Throughput is defined as the rate at which frames are successfully transmitted. For example, if in a one-hour period a channel was used for a full 45 minutes, and stood idle 15 minutes, the channel utilization would be equal to 75%. Now, if all packets transmitted could be scheduled perfectly with no collisions, the capacity of the channel would be 100%. Since all protocols implemented cannot prevent collisions from occurring, the maximum capacity of the channel falls below the 100% mark for perfect scheduling. Channel capacity formulas for various protocols [Kleinrock and Tobagi 1975] were implemented in the C language and are shown in Appendix I. Some example values are shown below:

Protocol	Capacity
Pure ALOHA	0.184
Slotted ALOHA	0.368
1-Persistent CSMA	0.529
Slotted 1-Persistent CSMA	0.531
0.1-Persistent CSMA	0.791
Nonpersistent CSMA	0.815
0.03-Persistent CSMA	0.827
Slotted Nonpersistent CSMA	0.857
Perfect Scheduling	1.000

These values are relevant for propagation delays of 0.01

milliseconds only. From the capacity of the network, operating below 100% efficiency, the throughput calculation is begun by multiplying the capacity by the baud rate of the network, i.e. 10 Mbps (million bits per second). This gives the maximum number of bits the protocol is capable of handling over the network in one second. If a user transmits packets of fixed length, then this fixed length is multiplied by the maximum number of bits the channel can handle within one second. If the user transmits X packets, then dividing X packets by the maximum number of packets the channel can handle gives the amount of time required to send those packets.

For instance, if a 1-persistent CSMA protocol is in use, the capacity is maximized at 52.9% efficiency as shown earlier. If the speed of the network is 10 Mbps, then the maximum number of bits that can be used in one second is:

0.529 * 10 Mbps = 5,290,000 bps

Assume a packet size of 1024 bits is used and that 25 packets need to be transmitted.

1024 bits/packet * 25 packets = 25,600 bits

25,600 bits / 5,290,000 bps = 0.004839 sec
= 4.839 ms

Therefore for a 1-persistent CSMA protocol, it requires a minimum of 4.839 milliseconds to send 25 packets of 1024 bits each assuming a propagation delay of 0.01 milliseconds.

But throughput is normally measured in units of packets per second. This can be achieved by multiplying the capacity of the network and the speed of the network together, and dividing the result by the packet size as shown below:

0.529 * 10 Mbps = 5,290,000 bps

5,290,000 bps / 1024 bits/packet = 5166.016 packets/sec

$$\frac{\text{bits}}{\text{sec}} / \frac{\text{bits}}{\text{packet}} = \frac{\text{bits}}{\text{sec}} * \frac{\text{packets}}{\text{bits}} = \frac{\text{packets}}{\text{sec}}$$

The formulas required for these calculations were also implemented in a C language program shown in Appendix J. For purposes of this project, it is the actual time required to send a certain number of packets that is of interest. This transmission time will be used in the network link of the simulation graph in the overall model of the distributed file system to represent the delay encountered by various types of networks. Certain factors such as propagation delay, offered traffic rate, persistence level (if applicable), packet size, number of packets, etc. will be varied to determine what affect they have on the system being simulated.

6. Results

6.1 Results of Simulation Runs

At the time of this writing the simulator is not completely functional. Although some statistics were generated for the verification test cases, no statistics were generated for any of the distributed file system models. This is due to the fact that all other models to be run have feedback loops within them which causes deadlock to occur. Deadlock detection and resolution has yet to be implemented effectively. From this, no determination can be made as to how well the Concurrent C simulator compares to output already obtained in [Hac 1986].

6.2 Validation of Project

Three discrete and seven continuous statistical distribution functions were implemented in the C language and then added to the Distributed Concurrent C simulator. To assure that each C function did indeed generate random numbers according to the associated distribution, a simple test program was devised. This test program simply generated 10000 numbers for a particular distribution, and then calculated a mean, variance, and standard deviation for the generated random values. The generated mean, and standard deviation were then compared to the mean and standard deviation supplied as input parameters to the function. This verified

that the function generated random numbers according to the distribution appropriately. This procedure was performed for each one of the ten distribution functions implemented. Each function appeared to produce random numbers appropriately according to the distribution with little error in the calculated mean and standard deviation. A listing of results is given in Table 6.1. As mentioned earlier, this code was implemented with additional tests to insure that no negative values were generated as service times. Upon further testing it was found that this restriction caused the average of the generated values to be slightly higher than usual.

The implementation of the formulas given for the distributed file system service times has been verified through inspection. Validation of the formulas representing the system being modeled has been rather difficult. Even though the formulas appear to represent the system, no validation has been completed since no derivation of these formulas was given in [Hac 1986].

The network capacity and throughput formulas [Kleinrock and Tobagi 1975] have been accepted as correct for over a decade. Their implementation into C code has been verified as correct through inspection. It is important to note that formulas with infinite summations have been approximated iteratively. The results for the capacity and throughput of each network appear to be acceptable.

Mean Values of Variates Generated for
Various Statistical Distributions

RANDOM - generates variates between 0 and ((2**31) - 1)
Average value = ((2**31) - 1) / 2 = 1073741823.5

Average of 10,000 variates

(A)	1076819980.9825	(C)	1073793989.0767
(B)	1067040692.9637	(D)	1083465318.6687

DRAND - generates variates between 0 and 1
Average value = 0.5

Average of 10,000 variates

(A)	0.503106	(C)	0.498616
(B)	0.496261	(D)	0.502270

UNIFORM

Input Values	Mean of 10,000 Variates Generated
1, 10	4.9836
1, 100	50.0823
-10, -1	-4.9549
50, 100	74.6208

BINOMIAL

Trials	Prob	Mean of 10,000 Variates Generated
25	0.25	6.2150
100	0.80	79.9542
50	0.44	21.9857
10	0.10	1.0073

Table 6.1

Mean Values of Variates Generated for
Various Statistical Distributions

POISSON Mean Input	Mean Generated	Mean Input	Mean Generated
50.0	49.6462	1.0	0.9961
5.0	5.0309	22.75	22.2894

EXPONENTIAL Mean Input	Mean Generated	Mean Input	Mean Generated
50.0	49.997475	1.0	0.986165
5.0	5.054825	22.75	22.435248

NORMAL Mean Input	St. Dev. Input	Mean Generated	St. Dev. Generated
50.0	3.5	50.021836	3.509798
5.0	1.4	5.006485	3.775254
122.6	34.7	122.139926	34.923966
100.0	10.0	99.864678	36.330613

LOGNORMAL Mean Input	St. Dev. Input	Mean Generated	St. Dev. Generated
50.0	3.5	50.027344	3.472112
5.0	1.4	4.981230	3.745457
122.6	34.7	122.866071	34.700001
100.0	10.0	99.995737	36.202155

Table 6.1 (cont.)

Mean Values of Variates Generated for
Various Statistical Distributions

ERLANG

Mean Input	Constant 'K'	Mean Generated
50.0	1	50.652954
5.0	1	5.050039
5.0	2	4.968532
1.0	1	0.993192

GAMMA

Mean Input	Constant 'K'	Mean Generated
50.0	1.0	49.490043
5.0	1.0	5.045628
25.0	8.0	25.054207
13.5	2.7	13.506715

WEIBULL

Shape	Scale	Mean Generated
1.0	50.0	50.980209
1.0	5.0	5.070828
1.0	1.0	0.996229
2.0	50.0	44.043974

BETA

K1	K2	Mean Generated	St. Dev. Generated
4.4	7.3	0.375488	0.136337
1.0	7.3	0.120910	0.201888
4.4	1.0	0.815246	0.252211
5.0	5.0	0.500839	0.293556

Table 6.1 (cont.)

Validation of the Distributed Concurrent C simulator was discussed in Chapter 3.

7. Conclusions and Future Research

Without results from the simulator, no conclusions can be drawn about the simulator's worthiness in simulating systems, or about the specific file system model discussed in this paper as it relates to this simulator. The Concurrent C simulator should provide results that would concur with conclusions drawn in [Hac 1986]. Specifically the fact that the CPU can become a bottleneck within the system. All jobs must be processed within the CPU before traveling to the next service center. This being the case, as the probability of delay increases, more jobs are executing in the delay nodes. This takes some of the load away from the CPU. As files are shared among transactions, these jobs spend even more time within the delay nodes. On the other hand, these delays due to locking of files can cause the system performance to decrease. System performance would be expected to increase if the locks were on records or sectors instead of entire files. The probability that any one transaction would require the same information in these former cases should be less than in latter case. As Hac states though, these claims can only be substantiated by further testing.

No conclusions can be made as to how different network protocols affect system performance in this model. It would be logical to assume that various time values inserted into the network link would be a factor in affecting system performance. It is difficult to determine whether this would

be a negative or positive affect. The number of jobs traveling across the network would be small compared to the number of jobs traveling to each of the other service centers within a host, simply because the probability of accessing another host is quite small. From this fact alone it would appear that it might require a large network delay to adversely affect system performance. This large network delay could come from the amount of data being transferred across the network, the propagation delay, the capacity of the network, etc. Each of these values play a role in determining the overall network delay. The point at which each parameter has an effect on system performance cannot be determined, however, without simulation results.

Many enhancements could be made to the Distributed Concurrent C simulator that would allow for even better simulations of the distributed file system model, as well as other models. The implementation of more complex nodes in a simulation graph, such as fission, fusion, and split nodes described in Chapter 2, could benefit the modeling of many systems, although they might not be beneficial to this particular file system model. These nodes would be implemented in the graphical front-end, and the semantics within the simulator itself, just as all other nodes are implemented. Grouping these and other nodes together into one node would allow large simulation graphs to be reduced into a group of subgraphs. An implementation of such a feature would only be done in the graphics front-end. The semantics of a

simulation graph would not change (to the simulator); only the aesthetics of the graph would change (to the user). This would allow for a more-detailed simulation of systems, such as the modeling of other aspects of a file system, or even a more-detailed breakdown of a network protocol.

Implementation of job variables would greatly expand the capabilities of the simulator. A job variable is simply an attribute of a job whose value could distinguish the job from all others. An example of such an attribute in the distributed file system model is the transaction class in which each transaction belongs. This attribute could be used to determine which path the job would take at a branch node. This would substantially reduce the number of paths and nodes required to model a system. Multiple probabilities could be assigned each individual path. Each probability would be "activated" only if a job possessing a certain value for an attribute arrived at that branch node. In other words with the use of job variables, the probabilities on each of the paths can vary as determined by the values of the attributes of the job. This would eliminate the need for many duplicate nodes providing the same service, as in the distributed file system model. The delays for various transaction classes were duplicated many times, once for each transaction class. With job variables only one delay need be implemented, as service times could vary depending on the value of the transaction class attribute to which the job belonged. Job variables are simply a more complex

construct to represent the same simulation graph as before that required many more arcs and nodes. The new graphs would appear cleaner as a result of this reduction. Many other characteristics of extended queueing networks supplemented to other simulators could be added here as well. At the time of this writing, design criteria is being developed to implement many of these features.

Regardless of the features provided by any simulator, its validity must be established. It would appear that more rigorous testing needs to be done to verify the simulator itself. Perhaps many more simulations need to be run, and the results compared to other established simulators. If possible some sort of proof method should be used in this verification to show that the simulator can indeed model any system realistically.

Further study of this particular model of a distributed file system might be undertaken. Other test cases might be run in which more than one file disk appeared on a host computer. Perhaps this system could be run on another simulator which can more accurately model the network protocol. Results could be compared between the two simulators to determine what affect the lower level modeling of the network might have on system performance.

References

Bulgren, W. G. 1982. Discrete System Simulation, Prentice-Hall, Inc., Englewood Cliffs, NJ.

Franta, W. R. and Chlamtac, I. 1981. Local Networks, D. H. Heath and Company, Lexington, Massachusetts.

Hac, A. 1986. "A Decomposition Solution to a Queueing Network Model of a Distributed File System with Dynamic Locking", IEEE TOSE SE-12,4 (April) 521-530.

Kleinrock, L. and Tobagi, F. 1975. "Packet Switching in Radio Channels: Part I - Carrier Sense Multiple-Access Modes and Their Throughput-Delay Characteristics", IEEE TCOM, COM-23,12 (December) 1400-1416.

Lavenberg, S. S., and Sauer, C. H. 1983. Computer Performance Modelling Handbook, Academic Press, New York.

Maekawa, M., Oldehoeft, A. E., and Oldehoeft, R. R. 1987. Operating Systems Advanced Concepts, The Benjamin/Cummings Publishing Company, Inc., Menlo Park, California.

Russell, E. C. 1983. Building Simulation Models with SIMSCRIPT II.5, CACI, Inc.-Federal, Los Angeles, California.

Sauer, C. H., and Chandy, K. M. 1981. Computer Systems Performance Modeling, Prentice-Hall, Inc., Englewood Cliffs, NJ.

Sauer, C. H., and MacNair, E. A. 1983. Simulation of Computer Communication Systems, Prentice-Hall, Inc., Englewood Cliffs, NJ.

Shotwell, R. E., ed. 1985. The Ethernet Sourcebook, Elsevier Science Publishing Company, Inc., New York.

Takagi, H. and Kleinrock, L. 1985. "Throughput Analysis for Persistent CSMA Systems", IEEE TCOM, COM-33,7 (July) 627-638.

Tanenbaum, A. S. 1981. Computer Networks, Prentice-Hall, Inc., Englewood Cliffs, NJ.

Appendix A

Implementation of the following three discrete and seven continuous distribution functions into the Concurrent C simulator was fairly straightforward.

Discrete	Continuous	
Uniform	Exponential	Erlang
Poisson	Normal	LogNormal
Binomial	Gamma	Weibull
	Beta	

Validation of input parameter values to each function was implemented within the graphical front-end of the simulator. This makes recovery more convenient. Should the user make an error in data entry, it can be corrected before the actual simulation is begun. The code to actually generate a number according to a specific distribution was implemented within the Concurrent C portion of the simulator. This code was supplemented with additional tests that assured (1) no division by zero errors could occur, and (2) that no negative values could be generated. A generated number less than zero represents a negative service time, or negative arrival time, which has no meaning. To prevent a negative time value from being used, another value is generated. This can cause the mean value of all of the generated values to be higher than usual. Implementation of these functions allows each server or source in the simulation graph to have its own service or arrival time generated from a particular distribution whose characteristics are entered into the

graphical front-end.

A.1. Discrete Distributions

Discrete distributions have the characteristic of producing only specific integer values. In other words in certain simulations, such as modeling the number of items purchased at a supermarket per person, it may not make sense to have a real (decimal) value being generated, i.e. 0.5. One-half of an item may not logically represent the actual system being modeled. Only integer values will suffice. The following three functions generate random integer values.

A uniform distribution is used to generate a specific integer value from a range of values. The minimum and maximum values in that range are supplied as input parameters to the function. The values entered for the range can be any positive or negative number. (Note: Care should be taken to assure that negative values also have appropriate meaning in the system being modeled.) Each integer value within the range then has an equal chance of being generated. The graph of a uniform distribution is shown in Figure A.1.

A Poisson distribution is usually used (1) to represent the number of arrivals within a certain period of time, or (2) to represent the number of occurrences of a specific

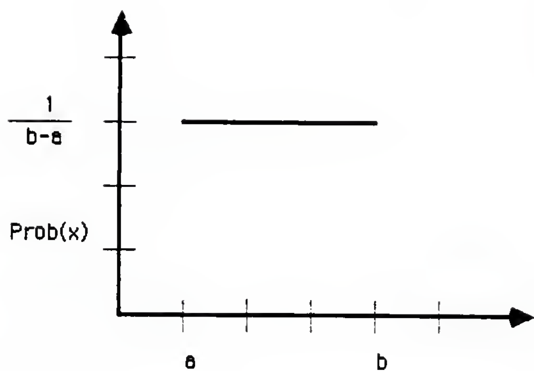


Figure A.1 - Uniform Distribution

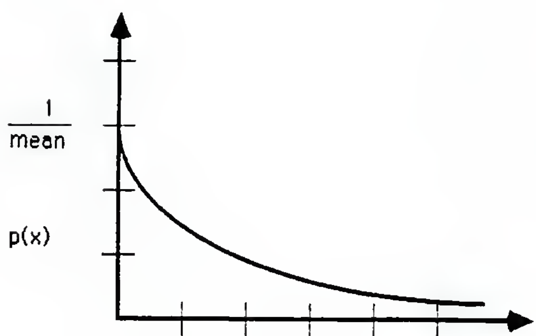


Figure A.2 - Exponential Distribution

event within a certain period of time. The mean number of arrivals or occurrences is provided to the function as an input parameter. Clearly this value must be greater than zero. An integer value is returned from the function representing a whole arrival or occurrence, whereas a partial arrival or occurrence would again be meaningless. Thus the Poisson function is also a discrete function.

A binomial distribution is used to represent the integer number of successes in 'n' independent trials, each having a probability of success 'p'. The classic example is that of rolling dice. If one die was thrown six times, how many times would the value of '1' appear? Each roll of the die is an independent trial, and each value on the die would have a one-sixth probability of success. For such simulations, a binomial distribution can model the system with accuracy. Again only integer values are produced for the number of successful occurrences, and the values for both 'p' and 'n' must be greater than zero.

A.2. Continuous Distributions

Continuous distributions, unlike discrete distributions, can produce any real (decimal) value. These values may be used for modeling such things as the average number of items purchased at a supermarket by a customer, service times, or arrival times. Any simulation in which real values have meaning, continuous distributions can be used.

The most commonly used continuous distribution is the exponential distribution. It is used to represent service times or arrival times. The exponential function can be related to the Poisson distribution if the number of arrivals in a specific time interval are Poisson-distributed. If this holds then the interarrival times are exponentially distributed. The mean value, around which the generated random numbers (or variates as they are called) are exponentially distributed, is supplied as an input parameter to the function. The variance of the generated values is equal to the square of the mean. This indicates that the generated values may be somewhat distant from the average as the value for the average increases. Other functions can partially alleviate this large variance by generating values closer to the mean. A graph of the exponential function is shown in Figure A.2.

The normal curve is familiar to most students as it usually shows the range of grades on an assignment. The graph of these points naturally produce a bell-shaped curve. The normal distribution function can generate random values that would comprise a bell curve. The mean and standard deviation of the values to be generated are entered as input parameters to the function. The mean can be less than zero, but the standard deviation, by definition, must be greater than zero. The normal function can generate negative values due to the "tail" of the function extending into the negative region of the x-axis. In generating service times, a

negative value would have no meaning. For other simulations, negative values may be appropriate. Again measures have been taken to assure that no negative values are ever generated in this implementation.

The gamma distribution is similar to the exponential function in that it is used in the generation of service times. The function requires a mean value and a constant 'k', both of which must be greater than zero, as input parameters. If this constant 'k' is set equal to one, then the gamma function is the same as the exponential function. If 'k' is an integer, then this function is the same as the erlang function (to be described shortly). Since the gamma function has a smaller variance in the random variates produced and has more control in the parameter selection ('k'), it realistically represents observed data somewhat more accurately than an exponential function. The gamma function is sometimes preferred over the exponential function in the generation of service times for this reason.

The beta distribution is closely related to the gamma distribution, but the resulting variates are restricted to values within the unit interval (0..1). Two constants, k1 and k2, are entered as input parameters to the function, both of which must be greater than zero. These two constants are used to describe the mean and standard deviation of the random numbers generated by the following two formulas:

```
mean = k1 / (k1+k2)
s.d. = sqrt[ (k1*k2) / (sqr[k1+k2] * [k1+k2+1]) ]
```

The erlang distribution is also closely related to the gamma distribution as mentioned earlier. The erlang function is a special case of the gamma function when the parameter 'k' in the gamma function is an integer. This function is also equivalent to the exponential function when 'k' is equal to one. Generation of service times is the main use for this function, as it is for its related functions.

The lognormal distribution is often used to characterize skewed data. The mean (m) and standard deviation (s) of the skewed data are entered as input parameters to the function. These two parameters are both restricted to values greater than zero. The mean (mu) and standard deviation (sd) of the values produced are described by the following equations:

```
mu = exp( m + [sqr(s) / 2] )
sd = exp( [m*2] + [sqr(s)] ) * ( [exp(sqr[s])] - 1 )
```

The weibull distribution is used to represent several families of distribution functions depending on the values of the two input parameters. The first parameter is called the shape parameter; the second is called the scale parameter. If the shape parameter is equal to one, then this function is the same as an exponential function with a mean that is equal to the scale parameter. if the shape parameter is set equal to two, there is a strong resemblance in

the weibull distribution and the gamma distribution.

For a clearer picture of all of the graphs of the functions just described and their associated formulas, see Chapter 4 of [Russell 1983].

Appendix B

The following C code segments represent the three discrete and seven continuous statistical distribution functions implemented for use by the Concurrent C simulator. These functions were ported from SIMSCRIPT II.5 as found in [Russell 1983] to the C language. A specialized random number generator used by the author of the Concurrent C simulator is shown first along with a type definition. Minor modifications were made to the code below to assure that (1) a zero value was not passed to a logarithmic function, (2) division by zero errors did not occur, and (3) no negative values were generated for service times. In the last case, another value is generated if the first value is equal to zero. These modifications do not appear below, but can be found in [Vopata 1988]. As mentioned earlier the input parameters validation is not done within this code, but within the graphics front-end [Butler 1989] of the simulator.

```
/*------------ RANDOM NUMBER GENERATOR [0,1] -------------*/
/*                                                        */

double drand01()
{
	double    p;
	extern    long    random();

	p = random() & 0xffffff;
	return p / 0xffffff;
}

/*                                                        */
/*--------------------------------------------------------*/


/*------------------ TYPE DEFINITION -------------------*/
/*                                                        */

typedef  double  real;

/*                                                        */
/*--------------------------------------------------------*/
```

```
/*************************************************************/
/*              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.                  */
/*                                                          */
/*----------------------------------------------------------*/

int uniform(a,b)
  int a,b;

{
  int c;

  if (a > b)
    {
      printf("\n In function uniform, Error:  a > b");
      printf("\n Lower bound must be < upper bound");
    }
  else
    {

      c = (int) (a + (b - a) * drand01());

      return (c);

    } /* else */

} /* end uniform */


/*---------- 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.                            */
/*                                                          */
/*----------------------------------------------------------*/

int poisson(mean)
  real mean;

{
```

```
  int n;
  real x,y;

  n = 0;

  if (mean > 6.0)
    return (normal(mean,sqrt(mean)));
  else
  if (mean <= 0.0)
    {
      printf("\n In function poisson, Error:  mean <= 0.0");
      printf("\n Mean must be greater than 0.0");
    }
  else
    {
      y = exp(-1 * mean);
      x = drand01();

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

        } /* while */

      return (n);

    } /* else */

}  /* end poisson */


/*---------- 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.                                 */
/*                                                        */
/*--------------------------------------------------------*/

int binomial(n,p)
  int n;
  real p;

{

  int i,sum = 0;

  if (n <= 0)
    {
      printf("\n In function binomial, Error:  n <= 0");
```

```
        printf("\n Number of trials must be greater than 0");
      }
    else
    if (p <= 0.0)
      {
        printf("\n In function binomial, Error:  p <= 0.0");
        printf("\n Prob. of occurrence must be > 0.0");
      }
    else
      {
        for (i=1; i <= n; ++i)
          {
            if (drand01() <= p)
              sum = sum + 1;
          }

        return (sum);

      } /* else */

  } /* end binomial */
```

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

/*--------- 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.        */
/*                                                        */
/*--------------------------------------------------------*/

real expntl(x)
  real x;

{

  if (x <= 0)
    {
      printf("\n In function exponential, Error: x <= 0");
      printf("\n Mean of function must be greater than 0");
    }
  else
    return ( (-x) * log(drand01()));

} /* expntl */


/*------------ 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.  */
/*                                                        */
/*--------------------------------------------------------*/

real normal(mu,sigma)
  real mu, sigma;

{

  real q,r,s,x,xx,y,yy;
```

```
  if (sigma <= 0.0)
    {
      printf("\n In function normal, Error: sigma <= 0.0");
      printf("\n Standard deviation must be > 0.0");
    }
  else
    {
      do {

        s = 2.0;

        while (s > 1.0)
          {
            x = drand01();
            y = (2 * drand01()) - 1;
            xx = x * x;
            yy = y * y;
            s = xx + yy;

         } /* while */

      r = sqrt( (-2) * log(drand01())) / s;
      q = r * sigma * (xx - yy) + mu;

      } /* do while */

      while (q <= 0.0);

    return (q);

} /* end normal */


/*------------ 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.        */
/*                                                         */
/*-------------------------------------------------------*/

real gamma(mean,k)
  real mean, k;

{

  real z,a,b,d,e,x,y,w;
  int kk,i;

  if (mean <= 0.0)
    {
      printf("\n In function gamma, Error: mean <= 0.0");
      printf("\n Mean must be greater than 0.0");
    }
  else
  if (k <= 0.0)
    {
      printf("\n In function gamma, Error: k <= 0.0");
      printf("\n K must be greater than 0.0");
    }
  else
    {
      z = 0.0;
      kk = (int) k;   /* truncation of k */
      d = k - kk;     /* fractional of k */

      if (kk != 0)
        {
          e = 1.0;

          for (i=1; i <= kk; ++i)
            e = e * drand01();

          z = -(log(e));

          if ((d == 0.0) && (k < 5.0))
            return((mean / k) * z);

        } /* end if */

      a = 1 / d;
      b = 1.0 / (1.0 - d);
      y = 2.0;

      while (y > 1.0)
        {
```

```
        x = pow(drand01()),a);
        y = (pow(drand01(),b)) + x;

      } /* while */

    w = x / y;
    y = -(log(drand01()));

    return ((w * y + z) * (mean / k));

  } /* else */

} /* end gamma */


/*------------ 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)  */
/*                                                         */
/*--------------------------------------------------------*/

real beta(k1,k2)
  real k1,k2;

{

  real x;

  if (k1 <= 0.0)
    {
      printf("\n In function beta, Error:  k1 <= 0.0");
      printf("\n K1 must be greater than 0.0");
    }
  else
  if (k2 <= 0.0)
    {
      printf("\n In function beta, Error:  k2 <= 0.0");
      printf("\n K2 must be greater than 0.0");
    }
  else
    {
      x = gamma(k1,k1);
      return (x / (x + gamma(k2,k2)));

    } /* else */

} /* end beta */
```

```
/*----------- 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.                                          */
/*                                                         */
/*---------------------------------------------------------*/

real erlang(x,k)
  real x;
  int k;

{

  int i;
  real e;

  if (x <= 0.0)
    {
      printf("\n In function erlang, Error: x <= 0.0");
      printf("\n The mean must be greater than 0.0");
    }
  else
  if (k <= 0)
    {
      printf("\n In function erlang, Error: k <= 0");
      printf("\n K must be greater than 0.0");
    }
  else
    {
      do {

        e = 1.0;

        for (i=1; i <= k; ++i)
          e = e * drand01();

      } /* do while */

      while (e == 0.0);

      return (-(x/k) * log(e));

    } /* else */

} /* end erlang */
```

```
/*--------- 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)                    */
/*                                                        */
/*--------------------------------------------------------*/

real lognormal(mean,sigma)
  real mean,sigma;

{

  real s,u;

  if (mean <= 0.0)
    {
      printf("\n In function lognormal, Error: mean <= 0.0");
      printf("\n Mean must be greater than 0.0");
    }
  else
  if (sigma <= 0.0)
    {
      printf("\n In function lognormal, Error: sigma<= 0.0");
      printf("\n Standard deviation must be > 0.0");
    }
  else
    {
      s = log((sigma * sigma) / (mean * mean) + 1);
      u = log(mean - (0.5 * s);

      return (exp(normal(u,sqrt(s))));

    } /* else */

} /* end lognormal */


/*----------- 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.                              */
/*                                                              */
/*-------------------------------------------------------------*/

real weibull(shape,scale)
  real shape,scale;

{

  if (shape <= 0.0)
    {
      printf("\n In weibull function, Error:  shape <= 0.0");
      printf("\n Shape parameter must be > 0.0");
    }
  else
  if (scale <= 0.0)
    {
      printf("\n In weibull function, Error:  scale <= 0.0");
      printf("\n Scale parameter must be > 0.0");
    }
  else
    return (scale * pow((-log(drand01())),(1.0 / shape)));

} /* end weibull */
```

Appendix C

The terms and formulas for calculating the service time of the delay components found in [Hac 1986] for the distributed file system model are shown below as they were decomposed for implementation. Notice the similarity between the formulas describing service times for a display and a terminal.

Let:

d-cpu[i]	delay experienced by transactions which require access to a file at a time when a transaction which locked the file is being executed in the CPU
d-disp[i]	delay experienced by transactions which require access to a file at a time when a transaction which locked the file is outputting characters on the display screen
d-term[i]	delay experienced by transactions which require access to a file at a time when a transaction which locked the file is at a terminal
t-cpu[i]	mean service time of CPU for a transaction of class i
t-disp[i]	mean service time of display output for a transaction of class i
t-term[i]	mean user "think" time for a transaction of class i
i	number of classes of transactions
num	number of transactions in the system
n[i]	number of transactions of class i
nc	number of classes of transactions which access files placed on different disks than the files accessed by transactions of class i and do not share files on these disks
p-disp[i]	prob. that a transaction of class i is sent to a display
p-term[i]	prob. that a transaction of class i is

```
                sent to a terminal

prob[i]         prob. of experiencing a delay when accessing
                a file (estimated as the ratio of the number
                of disk accesses made while the file is kept
                locked to the total number of disk accesses
                of a transaction of class i)

------------------------------------------------------------

d-cpu[i]        = min( max( term1, term2 ), term3 )
d-disp[i]       = min( max( term4, term5 ), term6 )
d-term[i]       = min( max( term7, term8 ), term9 )

------------------------------------------------------------

term4           =  term4a * (term4b - t-cpu[i] - term4c)

term4a          =  p[i] / (1 - p[i])
term4b          =  p-disp[i] * t-disp[i]
term4c          =  p-term[i] * (t-term[i] / (n[i] - 1))

term5           =  term5a * term5b

term5a          =  p-disp[i] * t-disp[i]
term5b          =  (n[i] - 1) / (n * (n - n[i]))

term6           =  t-disp[i]

------------------------------------------------------------

term7           =  term7a * (term7b - t-cpu[i] - term7c)

term7a          =  p[i] / (1 - p[i])
term7b          =  p-term[i] * t-term[i]
term7c          =  p-disp[i] * (t-disp[i] / (n[i] - 1))

term8           =  term8a * term8b

term8a          =  p-term[i] * t-term[i]
term8b          =  (n[i] - 1) / (n * (n - n[i]))

term9           =  t-term[i]

------------------------------------------------------------

term1           =  term1a * (summation - term1b - term1c)
term1a          =  (n - n[i]) * (p[i] / (1 - p[i]))
term1b          =  p-disp[i] * (t-disp[i] /
                      ((n[i] - 1) * (n - n[i])))
term1c          =  p-term[i] * (t-term[i] /
                      ((n[i] - 1) * (n - n[i])))

summation       =  summation = 0.0;
```

```
                    for (k=1; k<=nc; k++)
                      summation = summation + t-cpu[k];
                    summation = summation + t-cpu[i];

term2          =    summation * term2a
term2a         =    (n[i] - 1) / (n * (n - n[i]))

term3          =    summation
```

Appendix D

A description of each term and formula is given below for calculating the service times for both a display and a terminal in the model of a host computer taken from [Hac 1986]. These formulas represent the service times for the entire system and are not calculated for each transaction as were the formulas in the previous appendix.

Let:

s-disp	overall service time for display server in the model of the host
s-term	overall service time for terminal server in the model of the host
t-cpu[i]	mean service time of CPU for a transaction of class i
t-disp[i]	mean service time of display output for a transaction of class i
t-term[i]	mean user "think" time for a transaction of class i
i	number of classes of transactions
num	number of transactions in the system
n[i]	number of transactions of class i
nch	number of classes of transactions local to the host
p-disp[i]	prob. that a transaction of class i is sent to a display
p-term[i]	prob. that a transaction of class i is sent to a terminal

---

```
s-disp      = min( max( term10, term11 ), term12 )
s-term      = min( max( term13, term14 ), term15 )
```

---

s-disp (display service time for model of host)

```
term10 = (1/nch) * summation of
           (i=1 to nch) {term10a}

term10a = term10b - t-cpu[i] - term10c
term10b = p-disp[i] * t-disp[i]
term10c = p-term[i] * (t-term[i] / n[i])

term11 = (1/nch) * summation of
           (i=1 to nch) {term11a * term11b}

term11a = p-disp[i] * t-disp[i]
term11b = n[i] / (num * (num - n[i]))

term12 = (1/nch) * summation (i=1 to nch) {t-disp[i]}
```

```
s-term   (terminal service time for model of host)


term13 = (1/nch) * summation (i=1 to nch) {term13a}

term13a = term13b - t-cpu[i] - term13c
term13b = p-term[i] * t-term[i]
term13c = p-disp[i] * (t-disp[i] / n[i])

term14 = (1/nch) * summation of
           (i=1 to nch) {term14a * term14b}

term14a = p-term[i] * t-term[i]
term14b = n[i] / (num * (num - n[i]))

term15 = (1/nch) * summation (i=1 to nch) {t-term[i]}
```

Appendix E

```
/***********************************************************/
/*                                                        */
/*   Program:  Implementation of Hac's Formulas for       */
/*             Calculating Service Times for Delay        */
/*             Servers in the Simulation Model for        */
/*             a Distributed File System                  */
/*                                                        */
/*   Author:   Monte L. Hall                              */
/*                                                        */
/*   Purpose:  Masters Project Implementation             */
/*                                                        */
/*   This code is an attempt to implement the formulas    */
/*   for calculating service times for the CPU,           */
/*   Displays, and Terminals in the model of a delay for */
/*   the overall distributed file system model given by  */
/*   Hac.  It also calculates the service times for       */
/*   Displays and Terminals in the model of a host given */
/*   by Hac.                                              */
/*                                                        */
/***********************************************************/

#include <stdio.h>

#include <ctype.h>

     /* The following three definitions are declared */
     /* for use as array bounds.  They can be set to */
     /* any desired value as needed.                 */

#define  TRANS_MAX  10  /* MAX # of transactions classes */
#define  MAX_DISKS  10  /* MAX # of disks on any host */
#define  MAX_HOSTS  10  /* MAX # of hosts in
                           distributed system */

typedef double real;    /* a double is aliased as a real */

/*---------------------------------------------------------*/

real max(a,b)
  real a,b;
{
  if (a > b)
    return (a);
  else
    return (b);

} /* end max */

/*---------------------------------------------------------*/
```

```
real min(a,b)
  real a,b;
{
  if (a < b)
    return (a);
  else
    return (b);

} /* end min */

/*-------------------------------------------------------*/

main()
{

/*******************************************************/
/************* Declare all integer variables *************/
/*******************************************************/

  int   i;  /* used for loop counters */
  int   k;  /* used for loop counters */

  int   nh;    /* holds number of hosts declared by user */

  int   trans; /* holds number of transactions declared
                  by user */

  int   num;    /* holds total number of transactions
                   on the system */

  int   n[TRANS_MAX];  /* holds number of transactions of
                          a certain class; the class number
                          is the index into the array */

  int   nc[MAX_DISKS]; /* holds the number of classes of
                          transactions which do not access
                          files placed on this disk where
                          the disk number of a host is the
                          index into the array */

  int   ttl;    /* counts number of individual transactions
                   entered to allow a comparison to be made
                   against one parameter entered by the
                   the user representing the total number
                   of transactions on the system */

  int   ncd[MAX_HOSTS] [MAX_DISKS];
                /* holds the number of classes of
                   transactions accessing a disk server
                   where the host number and disk server
                   number are indices into the array */

  int   n_local[MAX_HOSTS];
                /* holds number of local transactions
```

```
                         on a host where the host number is
                         the index into the array */

int     nch[MAX_HOSTS];
                      /* holds number of local transaction
                         CLASSES on a host where the host
                         number is the index into the array */

int     disk_server[MAX_HOSTS];
                      /* holds number of disk servers on a
                         host where the host number is the
                         index into the array */

int     n_active[MAX_HOSTS] [TRANS_MAX];
                      /* holds number of active transactions
                         in a particular transaction class
                         on a particular host */


/**********************************************************/
/************** Declare all real variables ***************/
/**********************************************************/


real SUPERBIG;        /* used to represent infinity in
                         calculations in which a division
                         by zero error occurred */

real   c[TRANS_MAX];   /* holds temporary values in
                          calculation of pr_cpu */

real   n_remote[MAX_HOSTS] [MAX_HOSTS];
                     /* holds number of active transactions
                        on a host accessing another host */

real   ph[MAX_HOSTS];
                /* holds the total probability for a host that
                   it will be exited by transactions that are
                   executing temporarily on that host, that is
                   they are remote transactions; this is the
                   sum of all p_exit terms for each individual
                   transaction */

real   p_delay[TRANS_MAX];
                /* probability that a delay will occur for a
                   transaction accessing the disk where the
                   number of the transaction class is the index
                   into the array */

real   pr_cpu[TRANS_MAX];
              /* holds the calculated probability of accessing
                 the CPU after having accessed a disk server */

real   delay [TRANS_MAX];
```

```
                /* holds calculated probability of a delay
                   at a disk for a specific transaction */

real    no_delay [TRANS_MAX];
                /* holds calculated probability of no delay
                   at a disk for a specific transaction */

real    p_return[MAX_HOSTS] [MAX_HOSTS] [TRANS_MAX];
         /* holds the probability that a remote trans-
            action on host j returns to its local host k */

real    p_exit[TRANS_MAX];
            /* holds probability of a transaction class
               exiting a remote host as measured for
               input; this does not indicate the prob-
               ability of returning to a specific host
               as the previous variable declaration does */

real    p_ret[MAX_HOSTS];
                   /* holds probability of a transaction
                      returning to a particular host after
                      exiting another host */

real    s_cpu[TRANS_MAX];
                /* service time of CPU in model of a delay
                   for a class of transactions whose number
                   is the index into the array */

real    s_disp[TRANS_MAX];
                /* service time of display in model of a
                  delay for a class of transactions whose
                  number is the index into the array */

real    s_term[TRANS_MAX];
                /* service time of terminal in model of a
                   delay for a class of transactions whose
                   number is the index into the array */

real    t_cpu[TRANS_MAX];
                /* mean service time of CPU in model of a
                   host for a class of transactions whose
                   number is the index into the array */

real    t_disp[TRANS_MAX];
                /* mean service time of display in model of
                   a host for a class of transactions whose
                   number is the index into the array */

real    t_term[TRANS_MAX];
               /* mean service time of terminal in model of
                  a host for a class of transactions whose
                  number is the index into the array */

real    p_disp[TRANS_MAX];
```

```
             /* probability of accessing the display in
                the model of a host for a class of
                transactions whose number is the index
                into the array */

real  p_term[TRANS_MAX];
             /* probability of accessing the terminal in
                the model of a host for a class of
                transactions whose number is the index
                into the array */

real  p_disk[TRANS_MAX];
             /* probability of accessing the disk in the
                model of a host for a class of
                transactions whose number is the index
                into the array */

real  disp_server[MAX_HOSTS];
             /* holds calculated value of the overall
                service time of a display in the model
                of a host where the host number is the
                index into the array */

real  term_server[MAX_HOSTS];
             /* holds calculated value of the overall
                service time of a terminal in the model
                of a host where the host number is the
                index into the array */


             /* the following terms are used to hold
                temporary values and summation within
                the calculations that follow */

real  sum_a, sum_b, sum_c;

real  term1, term2, term3, term4, term5;
read  term6, term7, term8, term9;
real  term10, term11, term12, term13, term14, term15;
real  term1a, term1b, term1c;
real  term2a;
real  term4a, term4b, term4c;
real  term5a, term5b;
real  term7a, term7b, term7c;
real  term8a, term8b;
real  term10a, term10b, term10c;
real  term11a, term11b;
real  term13a, term13b, term13c;
real  term14a, term14b;
real  summation;

FILE  *fp, *fopen();

int a,b,d;
```

```
  fp = fopen("sim_data","w");

/********************************************************/
/****** Input all relevant data needed for formulas ******/
/********************************************************/

  printf("\nBegin input of user data\n");

  printf("\nPlease enter the number of hosts in the ");
  printf(" system:  ");
  scanf("%d",&nh);

  printf("\nPlease enter the number of TRANSACTION ");
  printf("CLASSES: ");
  scanf("%d",&trans);

  printf("\nPlease enter the TOTAL number of ");
  printf("transactions in the system:  ");
  scanf("%d",&num);
  printf("\n");

  ttl = 0;
  for (a=1; a<=trans; ++a)
    {
      printf("\nPlease enter the number of transaction");
      printf(" in class %d:  ",a);
      scanf("%d",&n[a]);
      ttl = ttl + n[a];
    }

  if (ttl != num)
    {
      printf("\nThe TOTAL number of transactions in the ");
      printf("system does NOT equal \nthe sum of the ");
      printf("individual number of transactions in ");
      printf("each class");
    }

  printf("\n");

  for (a=1; a<=nh; ++a)
    {
      printf("\nPlease enter the number of local ");
      printf("transactions on host %d:  ",a);
      scanf("%d",&n_local[a]);

      printf("\nPlease enter the number of transaction ");
      printf("CLASSES local to host %d:  ",a);
      scanf("%d",&nch[a]);

    } /* end for */

  for (a=1; a<=trans; ++a)
```

```
  {
    printf("\n\nFor all transactions in CLASS %d:",a);
    printf("\n\n  ---  SERVICE TIMES  --- ");

    printf("\n\n  The mean service time (in decimal ");
    printf("real) for the CPU is:  ");
    scanf("%f",&t_cpu[a]);

    printf("\n  The mean service time (in decimal ");
    printf("real) for a display output is:  ");
    scanf("%f",&t_disp[a]);

    printf("\n  The mean user 'think' time (decimal ");
    printf(" real) is:  ");
    scanf("%f",&t_term[a]);

    printf("\n\n  ---  PROBABILITIES  --- ");
    printf("\n\n  The probability (decimal real) that ");
    printf("a transaction\n  will be sent ");
    printf("to a display is:  ");
    scanf("%f",&p_disp[a]);

    printf("\n  The probability (decimal real) that a ");
    printf("transaction\n  will be sent ");
    printf("to a terminal is:  ");
    scanf("%f",&p_term[a]);

    printf("\n  The probability (decimal real) that a ");
    printf("transaction\n  will be sent to disk is:  ");
    scanf("%f",&p_disk[a]);

    printf("\n  The probability (decimal real) of ");
    printf("experiencing\n  a delay when accessing ");
    printf("a file is:  ");
    scanf("%f",&p_delay[a]);

    printf("\n  Please enter the number of classes of ");
    printf("transactions which access files\n  placed ");
    printf("on different disks than the files accessed ");
    printf("by transactions\n  of class %d, and that ",a);
    printf("do not share files on these disks:  ");
    scanf("%d",&nc[a]);

    printf("\n\nEnd Transaction Class %d\n",a);

  } /* end for */


for (a=1; a<=nh; ++a)
  {
    printf("\nHow many disk servers are on host %d:  ",a);
    scanf("%d",&disk_server[a]);

    for (b=1; b<=disk_server[a]; ++b)
```

```
      {
        printf("\nFor host %d and disk server %d",a,b);
        printf("\n  Please enter the number of classes ");
        printf("of transactions\n  accessing this ");
        printf("disk server:  ");
        scanf("%d",&ncd[a][b]);

      } /* end inner for */

  } /* end outer for */

i = 0;
for (a=1; a<=nh; ++a)
  {
    for (b=1; b<=nh; ++b)
       {
         if (a != b)
           {
             for (d=1; d<=nch[a]; ++d)
               {
                 printf("\nThe probability of transac");
                 printf("tion %d local to host %d",d+i,a);
                 printf("\n accessing host %d is:  ",b);
                 scanf("%f",&p_return[a][b][d]);

               } /* end for */

             i = i + d - 1;

           } /* end if */
       } /* end for */
   } /* end for */

i = 0;
if (nh > 1)
  {
    for (a=1; a<=nh; ++a)
      {
        for (b=1; b<=nch[a]; ++b)
          {
            printf("\nThe number of ACTIVE transactions");
            printf(" on host %d of class %d is:  ",a,b+i);
            scanf("%d",&n_active[a][b]);

          } /* end for */

       i = i + b - 1;

     } /* end for */


    for (a=1; a<=trans; ++a)
      {
```

```
        printf("\nThe probability that a transaction of");
        printf(" class %d\nexit a remote host is:  ",a);
        scanf("%f",&p_exit[a]);

      } /* end for */

    } /* end if */

  printf("\n\n**** End of User Input. ****\n\n");


/***********************************************************/
/******  Verification of User Input by Outputting It ******/
/***********************************************************/

  fprintf(fp,"\nThe following input has been recorded.");
  fprintf("fp,"  Please verify it.");
  fprintf(fp,"\n");

  fprintf(fp,"\nNumber of Hosts:                    %d",nh);
  fprintf(fp,"\nNumber of Transaction Classes:      %d",trans);

  for (a=1; a<=trans; ++a)
    {
      fprintf(fp,"\nNumber of Transactions in class");
      fprintf(fp," %d is:  %d",a,n[a]);
    }

  fprintf(fp,"\nTotal number of Transactions:        %d",num);
  fprintf(fp,"\n");

  for (a=1; a<=trans; ++a)
    {
      fprintf(fp,"\n\nTransactions of class %d:  \n",a);
      fprintf(fp,"\n  Display probability: ");
      fprintf(fp,"            %f",p_disp[a]);
      fprintf(fp,"\n  Terminal probability: ");
      fprintf(fp,"           %f",p_term[a]);
      fprintf(fp,"\n  Disk probability: ");
      fprintf(fp,"               %f",p_disk[a]);
      fprintf(fp,"\n  Delay probability: ");
      fprintf(fp,"              %f\n",p_delay[a]);
      fprintf(fp,"\n  Average CPU service time: ");
      fprintf(fp,"       %f",t_cpu[a]);
      fprintf(fp,"\n  Average Display service time: ");
      fprintf(fp,"  %f",t_disp[a]);
      fprintf(fp,"\n  Average Terminal service time: ");
      fprintf(fp," %f",t_term[a]);

    } /* end for */

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

  for (a=1; a<=nh; ++a)
```

```
  {
    fprintf(fp,"\nNumber of classes accessing host ");
    fprintf(fp,"%d is:          %d",a,nch[a]);
    fprintf(fp,"\nNumber of disk servers on host ");
    fprintf(fp," %d is:           %d",a,disk_server[a]);

    for (b=1; b<=disk_server[a]; ++b)
      {
        fprintf(fp,"\nNumber of classes accessing disk ");
        fprintf(fp,"server %d is:  %d",b,ncd[a][b]);
      }

    fprintf(fp,"\n");

  } /* end for */


i = 0;
for (a=1; a<=nh; ++a)
  {
    for (b=1; b<=nh; ++b)
       {
         if (a != b)
           {
             for (d=1; d<=nch[a]; ++d)
               {
                 fprintf(fp,"\nThe probability of trans");
                 fprintf(fp,"action %d local to",d+i);
                 fprintf(fp,"\nhost %d accessing host",a);
                 fprintf(fp," %d is:  ",b);
                 fprintf(fp,"%f",p_return[a][b][d]);

               } /* end for */

             i = i + d - 1;

           } /* end if */
       } /* end for */
   } /* end for */

fprintf(fp,"\n");

i = 0;
if (nh > 1)
  {
    for (a=1; a<=nh; ++a)
      {
        for (b=1; b<=nch[a]; ++b)
          {
            fprintf(fp,"\nThe number of ACTIVE trans");
            fprintf(fp,"actions on host %d of class",a);
            fprintf(fp," %d is:  %d",b+i,n_active[a][b]);

          } /* end for */
```

```
          i = i + b - 1;

        } /* end for */

      fprintf(fp,"\n");


      for (a=1; a<=trans; ++a)
        {
          fprintf(fp,"\nThe probability that a trans");
          fprintf(fp,"action of class %d\nexit a",a);
          fprintf(fp,"\nremote host is:  %f",p_exit[a]);

        } /* end for */

    } /* end if */

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


  SUPERBIG = 1.0 * (pow(2.0,32.0)); /* set infinity value */

/***********************************************************/
/******* Calculate total probability of transactions ******/
/*******                exiting each host           ******/
/***********************************************************/

  i = 0;

  for (a=1; a<=nh; ++a)
    {
      ph[a] = 0.0;

      for (b=1; b<=nch[a]; ++b)
        {
          ph[a] = ph[a] + p_exit[b+i];
        }

      i = b + i - 1;

    } /* end outer for */


/***********************************************************/
/***********  Calculate probability of delay at  **********/
/***********      disk for each transaction      **********/
/***********************************************************/

  for (i=1; i<=trans; ++i)
    {
      delay[i] = p_delay[i] * p_disk[i];
      no_delay[i] = (1 - p_delay[i]) * p_disk[i];

    } /* end for */
```

```
/*************************************************************/
/*********  Calculate service time for displays ***********/
/*********          in the model of a HOST       ***********/
/*************************************************************/

  b = 0;
  for (a=1; a<=nh; ++a)
    {

      sum_a = 0.0;
      sum_b = 0.0;
      sum_c = 0.0;

      for (i=1; i<=nch[a]; ++i)
        {

          term10b = p_disp[b+i] * t_disp[b+i];

          if (n[b+i] != 0)
            term10c = p_term[b+i] * (t_term[b+i] / n[b+i]);
          else
            {
              term10c = 0.0;
              printf("\nDivide by zero error occurred.  ");
              printf("Term was\nn[%d] in calculation",b+i);
              printf("\nof service time for display in ");
              printf("model of a host.\n");

           } /* end else */

          term10a = term10b - t_cpu[b+i] - term10c;

          sum_a = sum_a + term10a;

          term11a = p_disp[b+i] * t_disp[b+i];

          if ((num * (num - n[b+i])) != 0)
                     /* Divide by zero test */
            term11b = n[b+i]/(1.0 * (num * (num - n[b+i])));
          else
            {
              term11b = SUPERBIG;
              printf("\nDivide by zero error occurred.  ");
              printf("Term was\nnum * (num - n[%d]) ",b+i);
              printf("in calculation of service time for");
              printf("\ndisplay in model of a host.\n");

            } /* end else */

          sum_b = sum_b + (term11a * term11b);

          sum_c = sum_c + t_disp[b+i];

      } /* inner for */
```

```
    b = b + i - 1;

    term10 = sum_a / nch[a];
    term11 = sum_b / nch[a];
    term12 = sum_c / nch[a];

    disp_server[a] = min( max(term10,term11), term12 );

  } /* for */

/**********************************************************/
/*********  Calculate service time for terminals  *********/
/*********          in the model of a HOST         *********/
/**********************************************************/

  b = 0;
  for (a=1; a<=nh; ++a)
    {

      sum_a = 0.0;
      sum_b = 0.0;
      sum_c = 0.0;

      for (i=1; i<=nch[a]; ++i)
        {

          term13b = p_term[b+i] * t_term[b+i];

          if (n[b+i] != 0)
            term13c = p_disp[b+i] * (t_disp[b+i] / n[b+i]);
          else
            {
              term13c = 0.0;
              printf("\nDivide by zero error occurred.  ");
              printf("Term was\nn[%d] in calculation ",b+i);
              printf("\nof service time for terminal in ");
              printf("model of a host.\n");

            } /* end else */

          term13a = term13b - t_cpu[b+i] - term13c;

          sum_a = sum_a + term13a;

          term14a = p_term[b+i] * t_term[b+i];

          if ((num * (num - n[b+i])) != 0)
                      /* Divide by zero test */
            term14b = n[b+i]/(1.0 * (num * (num - n[b+i])));
          else
            {
              term14b = SUPERBIG;
              printf("\nDivide by zero error occurred.  ");
              printf("Term was\nnum * (num - n[%d])",b+i);
```

```
            printf(" in calculation of service time for");
            printf("\nterminal in model of a host.\n");

          }  /* end else */

        sum_b = sum_b + (term14a * term14b);

        sum_c = sum_c + t_term[b+i];

      } /* inner for */

    b = b + i - 1;

    term13 = sum_a / nch[a];
    term14 = sum_b / nch[a];
    term15 = sum_c / nch[a];

    term_server[a] = min( max(term13,term14), term15 );

  } /* for */


/***********************************************************/
/**********  Calculate service time for displays **********/
/**********        in the model of a DELAY       **********/
/***********************************************************/

  for (i=1; i<=trans; ++i)
    {

      term4a = p_delay[i] / (1 - p_delay[i]);
      term4b = p_disp[i] * t_disp[i];

      if ((n[i] - 1) != 0) /* Divide by zero test */
        term4c = p_term[i] * (t_term[i] / (n[i] - 1));
      else
        {
          term4c = SUPERBIG;
          printf("\nDivide by zero error occurred.  ");
          printf("Term was\n(n[%d] - 1) in calculation",i);
          printf("\nof service time for display in ");
          printf("model of a delay.\n");

        }  /* end else */

      term4  = term4a * (term4b - t_cpu[i] - term4c);

      term5a = p_disp[i] * t_disp[i];

      if ((num * (num - n[i])) != 0)
                /* Divide by zero test */
        term5b = 1.0 * (n[i] - 1) / (num * (num - n[i]));
      else
        {
```

```
        term5b = SUPERBIG;
        printf("\nDivide by zero error occurred.  ");
        printf("Term was\nnum * (num - n[%d]) in ",i);
        printf("calculation of service time for ");
        printf("\ndisplay in model of a delay.\n");

      }  /* end else */

    term5  = term5a * term5b;

    term6  = t_disp[i];

    s_disp[i] = min( max(term4,term5), term6 );

  } /* for */


/***********************************************************/
/*********  Calculate service time for terminals  *********/
/*********         in the model of a DELAY        *********/
/***********************************************************/


 for (i=1; i<=trans; ++i)
   {

    term7a = p_delay[i] / (1 - p_delay[i]);
    term7b = p_term[i] * t_term[i];

    if ((n[i] - 1) != 0) /* Divide by zero test */
      term7c = p_disp[i] * (t_disp[i] / (n[i] - 1));
    else
      {
        term7c = SUPERBIG;
        printf("\nDivide by zero error occurred.  ");
        printf("Term was\n(n[%d] - 1) in calculation ",i);
        printf("\nof service time for terminal in ");
        printf("model of a delay.\n");

      }  /* end else */

    term7  = term7a * (term7b - t_cpu[i] - term7c);

    term8a = p_term[i] * t_term[i];

    if ((num * (num - n[i])) != 0)
                /* Divide by zero test */
      term8b = 1.0 * (n[i] - 1) / (num * (num - n[i]));
    else
      {
        term8b = SUPERBIG;
        printf("\nDivide by zero error occurred.  ");
        printf("Term was\nnum * (num - n[%d]) in ",i);
        printf("calculation of service time for");
```

```
        printf("\nterminal in model of a delay.\n");

      }  /* end else */

    term8  = term8a * term8b;

    term9  = t_term[i];

    s_term[i] = min( max(term7,term8), term9 );

  } /* for */

/*************************************************************/
/**********  Calculate service time for the CPU ***********/
/**********       in the model of a DELAY        ***********/
/*************************************************************/

  for (i=1; i<=trans; ++i)
    {

    summation = 0.0;

    for (k=1; k<=nc[i]; ++k)
      summation = summation + t_cpu[k];

    summation = summation + t_cpu[i];

    term1a = (num - n[i]) * (p_delay[i]/(1 - p_delay[i]));
    if (((n[i] - 1) * (num - n[i])) != 0)
                      /* Divide by zero test */
      {
        term1b = p_disp[i] * (t_disp[i] /
                  ((n[i] - 1) * (num - n[i])));
        term1c = p_term[i] * (t_term[i] /
                  ((n[i] - 1) * (num - n[i])));
      }
    else
      {
        term1b = SUPERBIG;
        term1c = SUPERBIG;
        printf("\nDivide by zero error occurred.  Term");
        printf("was\n(n[%d] - 1) * (num - n[%d])",i,i);
        printf(" in calculation of service time for ");
        printf("\nCPU in model of a delay.\n");

      }  /* end else */

    term1  = term1a * (summation - term1b - term1c);

    if ((num * (num - n[i])) != 0)
                /* Divide by zero test */
      term2a = 1.0 * (n[i] - 1) / (num * (num - n[i]));
```

```
    else
      {
        term2a = SUPERBIG;
        printf("\nDivide by zero error occurred.  ");
        printf("Term was\nnum * (num - n[%d]) in ",i);
        printf("calculation of service time for ");
        printf("\nCPU in model of a delay.\n");

      }  /* end else */

    term2  = summation * term2a;

    term3  = summation;

    s_cpu[i] = min( max(term1,term2), term3 );

  } /* end for */


/************************************************************/
/*********  Calculate probability of accessing  ***********/
/*********        the CPU from a file disk       ***********/
/************************************************************/

  for (i=1; i<=trans; ++i)
    {
    if ((n[i] * (num - n[i])) != 0)
                /* Divide by zero test */
      {
        term1 = p_term[i] * (t_term[i] /
                   (n[i] * (num - n[i])));
        term2 = p_disp[i] * (t_disp[i] /
                   (n[i] * (num - n[i])));
      }
    else
      {
        term1 = SUPERBIG;
        term2 = SUPERBIG;
        printf("\nDivide by zero error occurred.  ");
        printf("Term was\nn[%d] * (num - n[%d]) in ",i,i);
        printf("calculation of the probability of ");
        printf("\naccessing the CPU from a file disk.\n");

      }  /* end else */

    if (p_disk[i] != 0.0)
      term3 = (1 - p_disk[i]) / p_disk[i];
    else
      {
        term3 = SUPERBIG;
        printf("\nDivide by zero error occurred.  ");
        printf("Term was\np_disk[%d] in calculation ",i);
        printf("\nof the probability of accessing the ");
        printf("CPU from a file disk.\n");
```

```
        }  /* end else */

      term4 = t_cpu[i] + term1 + term2;

      c[i]  = p_delay[i] * (t_cpu[i] + (term3 * term4));

    } /* end for */

  k = 0;

  for (a=1; a<=nh; ++a)  /* for every host */
    {
      for (b=1; b<=disk_server[a]; ++b)
                   /* for each disk server on that host */
        {

          summation = 0.0;

          for (d=1; d<=ncd[a][b]; ++d)
              /* for each transaction on that disk server */
            {
              summation = summation + n[k+d] / c[k+d];

            } /* end for */

          for (d=1; d<=ncd[a][b]; ++d)
            pr_cpu[k+d] = (n[k+d] / c[k+d]) / summation;

          k = k + d - 1;

        } /* end inner for */

    } /* end outer for */

/**************************************************************/
/********  Calculate number of active transactions   *******/
/********           accessing host j from k          *******/
/**************************************************************/

  for (a=1; a<=nh; ++a)
    {
      for (b=1; b<=nh; ++b)
        {
          if (a != b)
            {
              summation = 0.0;

              for (d=1; d<=nch[a]; ++d)
                summation = summation +
                   (p_return[a][b][d] * n_active[a][d]);

              n_remote[a][b] = summation;
```

```
          } /* end if */

        else
          n_remote[a][b] = 0.0;

      } /* end for */

    }  /* end for */

/***********************************************************/
/***  Calculate the probability of a remote transaction ***/
/***         returning to a particular host upon         ***/
/***               exiting the remote host               ***/
/***********************************************************/

  for (a=1; a<=nh; ++a)
    {
     summation = 0.0;

     for (b=1; b<=nh; ++b)
       {

        if (a != b)
          summation = summation + (n_active[a][b] * ph[a]);

       } /* end inner for */

     for (b=1; b<=nh; ++b)
       {

         if (a != b)
           p_ret[b] = (n_active[a][b] * ph[a]) / summation;

        } /* end inner for */

    } /* end outer for */

/***********************************************************/
/******  Print the results of the above calculations  *****/
/******               in tabular form                 *****/
/***********************************************************/

fprintf(fp,"\n************************************");
fprintf(fp,"*************");
fprintf(fp,"\n*  The following values have been ");
fprintf(fp,"calculated: *");
fprintf(fp,"\n************************************");
fprintf(fp,"*************\n\n");

for (i=1; i<=trans; ++i)
  {
   fprintf(fp,"\nProb. of delay at disk for transaction ");
```

```
    fprintf(fp,"class %d is:     %f",i,delay[i]);
    fprintf(fp,"\nProb. of NO delay at disk for trans");
    fprintf(fp,"action class %d is:  %f",i,no_delay[i]);
  }
fprintf(fp,"\n");

for (i=1; i<=trans; ++i)
  fprintf(fp,"\nCPU delay for class %d is:  %f",i,s_cpu[i]);
fprintf(fp,"\n");

for (i=1; i<=trans; ++i)
  {
    fprintf(fp,"\nDisplay delay for class %d is: ",i);
    fprintf(fp,"%f",s_disp[i]);
  }
fprintf(fp,"\n");

for (i=1; i<=trans; ++i)
  {
    fprintf(fp,"\nTerminal delay for class %d is: ",i);
    fprintf(fp,"%f",s_term[i]);
  }
fprintf(fp,"\n");

k = 0;
for (a=1; a<=nh; ++a)
  {
    for (b=1; b<=disk_server[a]; ++b)
      {
        for (i=1; i<=ncd[a][b]; ++i)
          {
            fprintf(fp,"\nProb of accessing CPU for class");
            fprintf(fp," %d is: %f",i+k,pr_cpu[i+k]);
          }

      } /* end inner for */

    k = k + i - 1;

  } /* end outer for */


fprintf(fp,"\n");
for (a=1; a<=nh; ++a)
  {
    fprintf(fp,"\nOverall service time of display server ");
    fprintf(fp,"on host %d is:   %f",a,disp_server[a]);

    fprintf(fp,"\nOverall service time of terminal server");
    fprintf(fp," on host %d is:  %f",a,term_server[a]);

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

```
for (a=1; a<=nh; ++a)
  {
    for (b=1; b<=nh; ++b)
      {
        if (a != b)
          {
            fprintf(fp,"\nNumber transactions on host ");
            fprintf(fp,"%d  accessing host %d is:  ",a,b);
            fprintf(fp,"%f",n_remote[a][b]);
          }

      } /* end for */

  } /* end for */

fprintf(fp,"\n");

for (a=1; a<=nh; ++a)
  {
    for (b=1; b<=nh; ++b)
      {
        if (a != b)
          {
            fprintf(fp,"\nProb of returning to host %d ",b);
            fprintf(fp,"from host %d \nby an exiting ",a);
            fprintf(fp,"transaction is:  %f",p_ret[b]);

          }  /* end if */

      } /* end for */

  } /* end for */

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

fclose(fp);

printf("\n\n\n*** All Calculations Completed! ***\n\n");

} /* end main */
```

Appendix F

This is the order in which the service times program will ask for input in a one-host system without file sharing. The data shown was used in one of the test cases.

Actual Data For Simulation of a 1-Host System (1st Run)
Probability of Delay = 0.1

(1) Number of Hosts: 1

(2) Number of Transaction Classes: 3

(3) Number of Transactions in Class 1: 2
Number of Transactions in Class 2: 2
Number of Transactions in Class 3: 2

(4) Number of Local Transaction Classes on Host 1: 3

(5) Number of Local Transactions on Host 1: 6

(6) Class 1: (service times in milliseconds)

(a) Mean CPU time: 42.0
(b) Mean Display time: 15.0
(c) Mean Terminal time: 1000.0

(d) Prob. of Accessing Display from CPU: 0.49
(e) Prob. of Accessing Terminal from CPU: 0.11
(f) Prob. of Accessing Disk from CPU: 0.4
(g) Prob. of Experiencing a Delay at Disk: 0.1

(h) Number Classes Accessing Other Disks than those Accessed by Class 1: 0

Class 2: (service times in milliseconds)

(a) Mean CPU time: 35.0
(b) Mean Display time: 25.0
(c) Mean Terminal time: 10000.0

(d) Prob. of Accessing Display from CPU: 0.19
(e) Prob. of Accessing Terminal from CPU: 0.01
(f) Prob. of Accessing Disk from CPU: 0.8
(g) Prob. of Experiencing a Delay at Disk: 0.1

(h) Number Classes Accessing Other Disks than

those Accessed by Class 1: 0

Class 3: (service times in milliseconds)

(a) Mean CPU time: 15.0
(b) Mean Display time: 15.0
(c) Mean Terminal time: 2000.0

(d) Prob. of Accessing Display from CPU: 0.61
(e) Prob. of Accessing Terminal from CPU: 0.09
(f) Prob. of Accessing Disk from CPU: 0.3
(g) Prob. of Experiencing a Delay at Disk: 0.1

(h) Number Classes Accessing Other Disks than those Accessed by Class 1: 0

(7) Number of File Disks on Host 1 = 1

(8) Number of Transaction Classes Accessing File Disk 1 = 3

Appendix G

This is the order in which the service times program will ask for input in the case of a two-host system with file-sharing.

Actual Data For Simulation of a 2-Host System (1st Run)
Probability of Delay = 0.1

---

(1) Number of Hosts: 2

(2) Number of Transaction Classes: 6

(3) Total Number of Transactions in the System: 12

(4) Number of Transactions in Class 1: 2
Number of Transactions in Class 2: 2
Number of Transactions in Class 3: 2
Number of Transactions in Class 4: 2
Number of Transactions in Class 5: 2
Number of Transactions in Class 6: 2

(5) Number of Local Transactions on Host 1: 6
Number of Local Transaction Classes on Host 1: 3
Number of Local Transactions on Host 2: 6
Number of Local Transaction Classes on Host 2: 3

(6) Host 1:

Class 1: (service times in milliseconds)

(a) Mean CPU time: 42.0
(b) Mean Display time: 15.0
(c) Mean Terminal time: 1000.0

(d) Prob. of Accessing Display from CPU: 0.47
(e) Prob. of Accessing Terminal from CPU: 0.10
(f) Prob. of Accessing Disk from CPU: 0.39
(g) Prob. of Experiencing a Delay at Disk: 0.1

(h) Number of Transaction Classes that do NOT Access File Disks that ARE Accessed by Transactions of this Class and do NOT Share Files on these Disks: 0

Class 2: (service times in milliseconds)

(a) Mean CPU time: 35.0
(b) Mean Display time: 25.0
(c) Mean Terminal time: 10000.0

(d) Prob. of Accessing Display from CPU: 0.17
(e) Prob. of Accessing Terminal from CPU: 0.02
(f) Prob. of Accessing Disk from CPU: 0.74
(g) Prob. of Experiencing a Delay at Disk: 0.1

(h) Number of Transaction Classes that do NOT Access File Disks that ARE Accessed by Transactions of this Class and do NOT Share Files on these Disks: 0

Class 3: (service times in milliseconds)

(a) Mean CPU time: 15.0
(b) Mean Display time: 15.0
(c) Mean Terminal time: 2000.0

(d) Prob. of Accessing Display from CPU: 0.59
(e) Prob. of Accessing Terminal from CPU: 0.09
(f) Prob. of Accessing Disk from CPU: 0.29
(g) Prob. of Experiencing a Delay at Disk: 0.1

(h) Number of Transaction Classes that do NOT Access File Disks that ARE Accessed by Transactions of this Class and do NOT Share Files on these Disks: 0

Host 2:

Class 4: (service times in milliseconds)

(a) Mean CPU time: 224.0
(b) Mean Display time: 20.0
(c) Mean Terminal time: 1000.0

(d) Prob. of Accessing Display from CPU: 0.19
(e) Prob. of Accessing Terminal from CPU: 0.04
(f) Prob. of Accessing Disk from CPU: 0.70
(g) Prob. of Experiencing a Delay at Disk: 0.1

(h) Number of Transaction Classes that do NOT Access File Disks that ARE Accessed by Transactions of this Class and do NOT Share Files on these Disks: 0

Class 5: (service times in milliseconds)

(a) Mean CPU time: 32.0
(b) Mean Display time: 15.0
(c) Mean Terminal time: 500.0

(d) Prob. of Accessing Display from CPU: 0.025
(e) Prob. of Accessing Terminal from CPU: 0.007
(f) Prob. of Accessing Disk from CPU: 0.88
(g) Prob. of Experiencing a Delay at Disk: 0.1

(h) Number of Transaction Classes that do NOT Access File Disks that ARE Accessed by Transactions of this Class and do NOT Share Files on these Disks: 0

Class 6: (service times in milliseconds)

(a) Mean CPU time: 15.0
(b) Mean Display time: 15.0
(c) Mean Terminal time: 2000.0

(d) Prob. of Accessing Display from CPU: 0.59
(e) Prob. of Accessing Terminal from CPU: 0.09
(f) Prob. of Accessing Disk from CPU: 0.29
(g) Prob. of Experiencing a Delay at Disk: 0.1

(h) Number of Transaction Classes that do NOT Access File Disks that ARE Accessed by Transactions of this Class and do NOT Share Files on these Disks: 0

(7) Number of File Disks on Host 1: 1
Number of Transaction Classes Accessing File Disk 1 on Host 1: 3
Number of File Disks on Host 2: 1
Number of Transaction Classes Accessing File Disk 1 on Host 2: 3

(8) Prob. of Transaction Class 1 Accessing Remote Host: 0.04
Prob. of Transaction Class 2 Accessing Remote Host: 0.07
Prob. of Transaction Class 3 Accessing Remote Host: 0.03
Prob. of Transaction Class 4 Accessing Remote Host: 0.07
Prob. of Transaction Class 5 Accessing Remote Host: 0.088
Prob. of Transaction Class 6 Accessing Remote Host: 0.03

(9) Number of ACTIVE Transactions on Host 1 of Class 1: 2
Number of ACTIVE Transactions on Host 1 of Class 2: 2
Number of ACTIVE Transactions

on Host 1 of Class 3: 2
Number of ACTIVE Transactions
on Host 2 of Class 4: 2
Number of ACTIVE Transactions
on Host 2 of Class 5: 2
Number of ACTIVE Transactions
on Host 2 of Class 6: 2

(10) Prob. of Transaction Class 1
Exiting Remote Host: 0.21
Prob. of Transaction Class 2
Exiting Remote Host: 0.013
Prob. of Transaction Class 3
Exiting Remote Host: 0.222
Prob. of Transaction Class 4
Exiting Remote Host: 0.057
Prob. of Transaction Class 5
Exiting Remote Host: 0.007
Prob. of Transaction Class 6
Exiting Remote Host: 0.222

Appendix H

The information required by the simulator program includes the type of each node (source, server-queue, branch, or sink), and an associated unique identifier for that node. If a statistical distribution is to be used for generating arrival times or service times, information regarding which distribution is to be used and appropriate parameters values are supplied. Each available machine was given a unique identifier also, and the user can specify on which machine each node in the simulation graph will execute. Sources must know the identifiers of nodes in which they feed into, while sinks must know the identifiers of nodes which feed into them. Server-queues must (1) have a queue-size, (2) have a queue discipline, (3) know how many nodes feed into the queue, and (4) know the identifier of the node into which the server feeds into. Probabilities are assigned to each branch coming out of a branch node, and the identifiers of outgoing nodes must be known. Finally the simulation stop-time must be entered along with the interval of time after which each group of simulation statistics are generated. Each parameter for each node is commented before the actual input record is given.

```
#   Begin Simulation Data for 1-Host System with Delay
#     Probability = 0.1
#   Comments refer to the line below it.
#
#
#   Queue/Server Type 2 (File Disk), Id 0, Distribution
#     Fixed 0, No min, No max, Fixed Time = 30.0 milli-
#     seconds, Run on Machine 12, Virtual Processor 0,
#     Feed into Node 4, Queue Size 15, FIFO Queue Discipline
#     0, Number Arcs coming in 6
#
(   (2 0) (0  0  0  30.0)  12  0  4  15  0  6 )
#
#   Source Type 0 (of Class 1 jobs), Id 1, Distribution
#     Fixed 0, No min, No max, Mean = 1 millisecond, Run on
#     Machine 12, VP 0, Generate 2 jobs, Feed into Node 4
#
(   (0 1) (0  0  0   1)  12  0  2  4)
#
#   Source Type 0 (of Class 2 jobs), Id 2, Distribution
#     Fixed 0, No min, No max, Mean = 1 millisecond, Run on
#     Machine 12, VP 0, Generate 2 jobs, Feed into Node 4
#
(   (0 2) (0  0  0   1)  12  0  2  4)
#
#   Source Type 0 (of Class 3 jobs), Id 3, Distribution
#     Fixed 0, No min, No max, Mean = 1 millisecond, Run on
#     Machine 12, VP 0, Generate 2 jobs, Feed into Node 4
#
(   (0 3) (0  0  0   1)  12  0  2  4)
#
#   Queue/Server Type 2 (CPU), Id 4, Distribution Exponential
#     4, No min, No max, Mean = 31 milliseconds, Run on
#     Machine 12, VP 0, Feed into Node 5, Queue Size 15,
#     FIFO Queue Discipline 0, Number Arcs coming in 6
#
(   (2 4) (4  0  0  31)  12  0  5  15  0  6)
#
#   Branch Type 3 (Which Transaction Class), Id 5, Run on
#     Machine 12, VP 0, Number Arcs coming in 1,
#     Number Arcs going out 3,  Branch to Node 6 with
#     probability 1/3, Branch to Node 7 with probability
#     1/3, Branch to Node 8 with probability 1/3
#
(   (3 5) 12 0 1 3  ((6 0.333333) (7 0.333333) (8 0.333333)))
#
#   Branch Type 3 (Class 3 Going Where), Id 6, Run on
#     Machine 12, VP 0, Number Arcs coming in 1,
#     Number Arcs going out 3,  Branch to Node 9 with
#     probability 0.61, Branch to Node 10 with probability
#     0.09, Branch to Node 11 with probability 0.3
#
(   (3 6)  12  0  1  3  ( (9  0.61) (10  0.09) (11  0.3) )  )
```

```
#
#   Branch Type 3 (Class 2 Going Where), Id 7, Run on
#     Machine 12, VP 0, Number Arcs coming in 1,
#     Number Arcs going out 3,  Branch to Node 9 with
#     probability 0.19, Branch to Node 10 with probability
#     0.01, Branch to Node 16 with probability 0.8
#
(   (3 7)  12  0  1  3  ( (9  0.19) (10  0.01) (16  0.8) )  )
#
#   Branch Type 3 (Class 1 Going Where), Id 8, Run on
#     Machine 12, VP 0, Number Arcs coming in 1,
#     Number Arcs going out 3,  Branch to Node 9 with
#     probability 0.49, Branch to Node 10 with probability
#     0.11, Branch to Node 21 with probability 0.4
#
(   (3 8)  12  0  1  3  ( (9  0.49) (10  0.11) (21  0.4) )  )
#
#   Queue/Server Type 2 (Display), Id 9, Distribution Fixed
#     0, No min, No max, Fixed Time = 0.590278 milliseconds,
#     Run on Machine 12, VP 0, Feed into Node 4,
#     Queue Size 15, FIFO Queue Discipline 0,
#     Number Arcs coming in 3
#
(   (2 9) (0  0  0  0.590278)  12  0  4  15  0  3 )
#
#   Queue/Server Type 2 (Terminal), Id 10, Distribution Fixed
#     0, No min, No max, Fixed Time = 95.791668 milliseconds,
#     Run on Machine 12, VP 0, Feed into Node 4,
#     Queue Size 15, FIFO Queue Discipline 0,
#     Number Arcs coming in 3
#
(   (2 10) (0  0  0  95.791668)  12  0  4  15  0  3 )
#
#   Branch Type 3 (Class 3 Delay or Not), Id 11, Run on
#     Machine 12, VP 0, Number Arcs coming in 1,
#     Number Arcs going out 2,  Branch to Node 0 with
#     probability 0.9, Branch to Node 12 with probability 0.1
#
(   (3 11)  12  0  1  2  ( (0  0.9)  (12  0.1) ) )
#
#   Queue/Server Type 2 (Class 3 CPU Delay), Id 12,
#     Distribution Fixed 0, No min, No max, Fixed Time =
#     0.625 milliseconds, Run on Machine 12, VP 0,
#     Feed into Node 13, Queue Size 15, FIFO Queue Discipline
#     0, Number Arcs coming in 3
#
(   (2 12)  (0  0  0  0.625)  12  0  13  15  0  3 )
#
#   Branch Type 3 (Class 3 Continue to Delay or Not), Id 13,
#     Run on Machine 12, VP 0, Number Arcs coming in 1,
#     Number Arcs going out 3,  Branch to Node 0 with
#     probability 0.3, Branch to Node 14 with probability
#     0.61, Branch to Node 15 with probability 0.09
#
```

```
(   (3 13)  12  0  1  3  ( (0  0.3) (14  0.61) (15  0.09) ) )
#
#   Queue/Server Type 2 (Class 3 Display Delay), Id 14,
#     Distribution Fixed 0, No min, No max, Fixed Time =
#     0.381250 milliseconds, Run on Machine 12, VP 0,
#     Feed into Node 12, Queue Size 15, FIFO Queue Discipline
#     0, Number Arcs coming in 1
#
(   (2 14) (0  0  0  0.381250)  12  0  12  15  0  1 )
#
#   Queue/Server Type 2 (Class 3 Terminal Delay), Id 15,
#     Distribution Fixed 0, No min, No max, Fixed Time =
#     17.316668 milliseconds, Run on Machine 12, VP 0,
#     Feed into Node 12, Queue Size 10, FIFO Queue Discipline
#     0, Number Arcs coming in 1
#
(   (2 15) (0  0  0  17.316668)  12  0  12  10  0  1 )
#
#   Branch Type 3 (Class 2 Delay or Not), Id 16, Run on
#     Machine 12, VP 0, Number Arcs coming in 1,
#     Number Arcs going out 2,  Branch to Node 0 with
#     probability 0.9, Branch to Node 17 with probability 0.1
#
(   (3 16)  12  0  1  2  ( (0  0.9)  (17  0.1) ) )
#
#   Queue/Server Type 2 (Class 2 CPU Delay), Id 17,
#     Distribution Fixed 0, No min, No max, Fixed Time =
#     3.916667 milliseconds, Run on Machine 12, VP 0,
#     Feed into Node 18, Queue Size 15, FIFO Queue Discipline
#     0, Number Arcs coming in 3
#
(   (2 17) (0  0  0  3.916667)  12  0  18  15  0  3 )
#
#   Branch Type 3 (Class 2 Continue to Delay or Not), Id 18,
#     Run on Machine 12, VP 0, Number Arcs coming in 1,
#     Number Arcs going out 3,  Branch to Node 0 with
#     probability 0.8, Branch to Node 19 with probability
#     0.19, Branch to Node 20 with probability 0.01
#
(   (3 18)  12  0  1  3  ( (0  0.8) (19  0.19) (20  0.01) ) )
#
#   Queue/Server Type 2 (Class 2 Display Delay), Id 19,
#     Distribution Fixed 0, No min, No max, Fixed Time =
#     0.197917 milliseconds, Run on Machine 12, VP 0,
#     Feed into Node 17, Queue Size 15, FIFO Queue Discipline
#     0, Number Arcs coming in 1
#
(   (2 19) (0  0  0  0.197917)  12  0  17  15  0  1 )
#
#   Queue/Server Type 2 (Class 2 Terminal Delay), Id 20,
#     Distribution Fixed 0, No min, No max, Fixed Time =
#     6.694444 milliseconds, Run on Machine 12, VP 0,
#     Feed into Node 17, Queue Size 15, FIFO Queue Discipline
#     0, Number Arcs coming in 1
```

```
#
(   (2 20) (0  0  0  6.694444)  12  0  17  15  0  1 )
#
#   Branch Type 3 (Class 1 Delay or Not), Id 21, Run on
#     Machine 12, VP 0, Number Arcs coming in 1,
#     Number Arcs going out 2,  Branch to Node 0 with
#     probability 0.9, Branch to Node 22 with probability 0.1
#
(   (3 21)  12  0  1  2  ( (0  0.9)  (22  0.1) ) )
#
#   Queue/Server Type 2 (Class 1 CPU Delay), Id 22,
#     Distribution Fixed 0, No min, No max, Fixed Time =
#     5.627778 milliseconds, Run on Machine 12, VP 0,
#     Feed into Node 23, Queue Size 15, FIFO Queue Discipline
#     0, Number Arcs coming in 3
#
(   (2 22) (0  0  0  5.627778)  12  0  23  15  0  3 )
#
#   Branch Type 3 (Class 1 Continue to Delay or Not), Id 23,
#     Run on Machine 12, VP 0, Number Arcs coming in 1,
#     Number Arcs going out 3,  Branch to Node 0 with
#     probability 0.4, Branch to Node 24 with probability
#     0.49, Branch to Node 25 with probability 0.11
#
(   (3 23)  12  0  1  3  ( (0  0.4) (24  0.49) (25  0.11) ) )
#
#   Queue/Server Type 2 (Class 1 Display Delay), Id 24,
#     Distribution Fixed 0, No min, No max, Fixed Time =
#     0.306250 milliseconds, Run on Machine 12, VP 0,
#     Feed into Node 22, Queue Size 15, FIFO Queue Discipline
#     0, Number Arcs coming in 1
#
(   (2 24) (0  0  0  0.306250)  12  0  22  15  0  1 )
#
#   Queue/Server Type 2 (Class 1 Terminal Delay), Id 25,
#     Distribution Fixed 0, No min, No max, Fixed Time =
#     6.738889 milliseconds, Run on Machine 12, VP 0,
#     Feed into Node 22, Queue Size 15, FIFO Queue Discipline
#     0, Number Arcs coming in 1
#
(   (2 25) (0  0  0  6.738889)  12  0  22  15  0  1 )
#
#
#   End of Simulation Data for 1-Host System with
#     Delay Probability = 0.1
```

Appendix I

```
/**************************************************************/
/*                                                            */
/*  Program:  Implementation of Network                       */
/*              Capacity Formulas                             */
/*                                                            */
/*  Author:   Monte L. Hall                                   */
/*                                                            */
/*  Purpose:  Masters Project Implementation                  */
/*                                                            */
/*    This program is an attempt to implement some of the     */
/*    formulas given by Kleinrock, Tobagi, and Takagi.        */
/*    These formulas give the capacity of a specific          */
/*    network given input parameters such as offered          */
/*    traffic rate, propagation delay, persistence level      */
/*    (if applicable), etc.  Each formula implemented         */
/*    (as a function) has a reference as to where the        */
/*    author found the formula in publication.  The           */
/*    notation used for function names is as follows:         */
/*                                                            */
/*      i -> infinite users, no i -> finite users             */
/*      s -> slotted, no s -> unslotted                       */
/*      1 -> 1-persistent, p -> p-persistent                  */
/*                                                            */
/**************************************************************/

#include <stdio.h>

#include <ctype.h>

#include <math.h>

#define ITERATIONS 2000
#define LINELENGTH   80


/*********** PURE ALOHA with INFINITE USERS ***************/
/*                                                            */
/*  Reference:                                                */
/*    Kleinrock, Leonard, and Fouad A. Tobagi, Packet         */
/*      Switching in Radio Channels:  Part I - Carrier        */
/*      Sense Multiple-Access Modes and Their Throughput-     */
/*      Delay Characteristics, IEEE Transactions on           */
/*      Communications, Volume COM-23, no. 12, December       */
/*      1975, p.1403.                                         */
/*                                                            */
/**************************************************************/

i_pure_aloha()
{
```

```
double s = 0.0, G = 0.0;

  printf("\n\nWhat is the offered traffic rate?");
  printf(" (in decimal real)\n");
  gets(line);
  sscanf(line,"%e",&G);

  while (G < 0.0)
    {
      printf("\nThe offered traffic rate must be greater");
      printf(" than 0.0");
      printf("\nWhat is the offered traffic rate?");
      printf(" (in decimal real)\n");
      gets(line);
      sscanf(line,"%e",&G);

    } /* end while */

  s = G * exp(-2.0 * G);

  printf("\nAt an offered rate(packets/transmission-time)");
  printf(" of:  %f",G);
  printf("\nthe throughput of the network is:  %f",s);
  printf("0);

  printf("\nThe maximum throughput possible is 0.184 or");
  printf(" 18 percent efficiency");
  printf("\n");

  return(1);

}  /* end PURE ALOHA with INFINITE USERS */


/********** SLOTTED ALOHA with INFINITE USERS *************/
/*                                                        */
/*  Reference:                                            */
/*    Kleinrock, Leonard, and Fouad A. Tobagi, Packet     */
/*      Switching in Radio Channels:  Part I - Carrier    */
/*      Sense Multiple-Access Modes and Their Throughput- */
/*      Delay Characteristics, IEEE Transactions on       */
/*      Communications, Volume COM-23, no. 12, December   */
/*      1975, p. 1403.                                    */
/*                                                        */
/**********************************************************/


i_slotted_aloha()
{

double G = 0.0, s = 0.0;

  printf("\n\nWhat is the offered traffic rate?");
  printf(" (in decimal real)\n");
```

```
  gets(line);
  sscanf(line,"%f",&G);

  while (G < 0.0)
    {
      printf("\nThe offered traffic rate must be greater");
      printf(" than 0.0");
      printf("\nWhat is the offered traffic rate?");
      printf(" (in decimal real)\n");
      gets(line);
      sscanf(line,"%e",&G);

    } /* end while */

  s = G * (exp(-G));

  printf("\nAt an offered rate(packets/transmission-time)");
  printf(" of:  %f",G);
  printf("\nthe throughput of the network is:  %f",s);
  printf("\n");

  printf("\nThe maximum throughput possible is 0.368 or");
  printf(" 36.8 percent efficiency");
  printf("\n");

  return (1);

}  /* end SLOTTED ALOHA with INFINITE USERS */


/********* NONPERSISTENT with INFINITE USERS **************/
/*                                                         */
/*  Reference:                                             */
/*    Kleinrock, Leonard, and Fouad A. Tobagi, Packet      */
/*      Switching in Radio Channels:  Part I - Carrier     */
/*      Sense Multiple-Access Modes and Their Throughput-  */
/*      Delay Characteristics, IEEE Transactions on        */
/*      Communications, Volume COM-23, no. 12, December    */
/*      1975, p.1404.                                      */
/*                                                         */
/***********************************************************/


i_nonpersistent()
{

double a = 0.0, G = 0.0, s = 0.0;
double term1, term2;


  printf("\n\nWhat is the offered traffic rate?");
  printf(" (in decimal real)\n");
  gets(line);
  sscanf(line,"%e",&G);
```

```
  while (G < 0.0)
    {
      printf("\nThe offered traffic rate must be greater");
      printf(" than 0.0");
      printf("\nWhat is the offered traffic rate?");
      printf(" (in decimal real)\n");
      gets(line);
      sscanf(line,"%e",&G);

    } /* end while */


  printf("\n\nWhat is the propagation delay of the");
  printf(" network? (in decimal real)\n");
  gets(line);
  sscanf(line,"%e",&a);

  while (a < 0.0)
    {
      printf("\nThe propagation delay must be greater");
      printf(" than 0.0");
      printf("\n\nWhat is the propagation delay of the");
      printf(" network? (in decimal real)\n");
      gets(line);
      sscanf(line,"%e",&a);

    } /* end while */

  term1 = G * exp(-a * G);
  term2 = (G * (1.0 + (2.0*a))) + (exp(-a * G));
  s = term1 / term2;

  printf("\nAt an offered rate(packets/transmission-time)");
  printf(" of:  %f",G);
  printf("\nwith propagation delay of:  %f",a);
  printf("\nthe throughput of the network is:  %f",s);
  printf("\n\n");

  return (1);

}  /* end NONPERSISTENT with INFINITE USERS */


/****** SLOTTED NONPERSISTENT with INFINITE USERS *********/
/*                                                        */
/*  Reference:                                            */
/*    Kleinrock, Leonard, and Fouad A. Tobagi, Packet     */
/*      Switching in Radio Channels:  Part I - Carrier    */
/*      Sense Multiple-Access Modes and Their Throughput- */
/*      Delay Characteristics, IEEE Transactions on       */
/*      Communications, Volume COM-23, no. 12, December   */
/*      1975, p.1404.                                     */
/*                                                        */
/**********************************************************/
```

```
i_s_nonpersistent()
{

double a = 0.0, G = 0.0, s = 0.0;
double term1, term2;


  printf("\n\nWhat is the offered traffic rate?");
  printf(" (in decimal real)\n");
  gets(line);
  sscanf(line,"%e",&G);

  while (G < 0.0)
    {
      printf("\nThe offered traffic rate must be greater");
      printf(" than 0.0");
      printf("\nWhat is the offered traffic rate?");
      printf(" (in decimal real)\n");
      gets(line);
      sscanf(line,"%e",&G);

    } /* end while */

  printf("\n\nWhat is the propagation delay of the");
  printf(" network? (in decimal real)\n");
  gets(line);
  sscanf(line,"%e",&a);

  while (a < 0.0)
    {
      printf("\nThe propagation delay must be greater");
      printf(" than 0.0");
      printf("\n\nWhat is the propagation delay of the");
      printf(" network? (in decimal real)\n");
      gets(line);
      sscanf(line,"%e",&a);

    } /* end while */

  if (a == 0.0)
    {
      printf("\nA propagation delay of zero causes a");
      printf(" division by zero error.");
      printf("\nThis is equivalent to a throughput equal");
      printf(" to infinity.");

    } /* end if */
  else
    {
      term1 = a * (G * exp(-a * G));
      term2 = (1.0 - (exp(-a * G))) + a;
      s = term1 / term2;

      printf("\nAt an offered rate");
```

```
      printf(" (packets/transmission-time) of:  %f",G);
      printf("\nand a propagation delay of:  %f",a);
      printf("\nthe throughput of the network is:  %f",s);

    } /* end else */

  printf("\n\n");

  return (1);

}  /* end SLOTTED NONPERSISTENT with INFINITE USERS */


/******** 1-PERSISTENT CSMA with INFINITE USERS ***********/
/*                                                         */
/*  Reference:                                             */
/*    Kleinrock, Leonard, and Fouad A. Tobagi, Packet      */
/*      Switching in Radio Channels:  Part I - Carrier     */
/*      Sense Multiple-Access Modes and Their Throughput-  */
/*      Delay Characteristics, IEEE Transactions on        */
/*      Communications, Volume COM-23, no. 12, December    */
/*      1975, p.1404.                                      */
/*                                                         */
/*  Secondary Reference:                                   */
/*    Takagi, Hideaki, and Leonard Kleinrock, Throughput   */
/*      Analysis of Persistent CSMA Systems, IEEE          */
/*      Transactions on Communications, Volume COM-33      */
/*      no. 7, July, 1985, p. 634.                         */
/*                                                         */
/*                                                         */
/***********************************************************/


i_1_persistent_csma()
{

double term1, term2, term3, term4, term5, num, denom;
double a = 0.0, G = 0.0, s = 0.0;


  printf("\n\nWhat is the offered traffic rate?");
  printf(" (in decimal real)\n");
  gets(line);
  sscanf(line,"%e",&G);

  while (G < 0.0)
    {
      printf("\nThe offered traffic rate must be greater");
      printf(" than 0.0");
      printf("\nWhat is the offered traffic rate?");
      printf(" (in decimal real)\n");
      gets(line);
      sscanf(line,"%e",&G);
```

```
    } /* end while */

  printf("\n\nWhat is the propagation delay of the");
  printf(" network? (in decimal real)\n");
  gets(line);
  sscanf(line,"%e",&a);

  while (a < 0.0)
    {
      printf("\nThe propagation delay must be greater");
      printf(" than 0.0");
      printf("\n\nWhat is the propagation delay of the");
      printf(" network? (in decimal real)\n");
      gets(line);
      sscanf(line,"%e",&a);

    } /* end while */

  term1 = term2 = term3 = term4 = term5 = 0.0;
  num = denom = 0.0;

  term1 = G * (1.0 + G + ((a*G) * (1.0 + G + (a*G/2.0))));
  term2 = 1.0 * exp (-G * (1.0 + (2.0*a)));
  num = term1 * term2;

  term3 = G * (1.0 + (2.0*a));
  term4 = 1.0 - (exp (-a * G));
  term5 = (1.0 + (a*G)) * exp (-G * (1.0 + a));
  denom = term3 - term4 + term5;

  s = num / denom;

  printf("\nAt an offered rate(packets/transmission-time)");
  printf(" of:  %f",G);
  printf("\nwith a propagation delay of:  %f",a);
  printf("\nthe throughput of the network is:  %f",s);
  printf("\n\n");

  return(1);

}  /* end 1-PERSISTENT CSMA with INFINITE USERS */


/***** SLOTTED 1-PERSISTENT CSMA with INFINITE USERS ******/
/*                                                        */
/*  Reference:                                            */
/*    Kleinrock, Leonard, and Fouad A. Tobagi, Packet     */
/*      Switching in Radio Channels:  Part I - Carrier    */
/*      Sense Multiple-Access Modes and Their Throughput- */
/*      Delay Characteristics, IEEE Transactions on       */
/*      Communications, Volume COM-23, no. 12, December   */
/*      1975, p.1406.                                     */
/*                                                        */
/*  Secondary Reference:                                  */
```

```
/*    Takagi, Hideaki, and Leonard Kleinrock, Throughput   */
/*      Analysis of Persistent CSMA Systems, IEEE          */
/*      Transactions on Communications, Volume COM-33      */
/*      no. 7, July, 1985, p. 631.                         */
/*                                                         */
/***********************************************************/


i_s_1_persistent_csma()
{

double term1, term2, term3, term4, num, denom;
double a = 0.0, G = 0.0, s = 0.0;


  printf("\n\nWhat is the offered traffic rate?");
  printf(" (in decimal real)\n");
  gets(line);
  sscanf(line,"%e",&G);

  while (G < 0.0)
    {
      printf("\nThe offered traffic rate must be greater");
      printf(" than 0.0");
      printf("\nWhat is the offered traffic rate?");
      printf(" (in decimal real)\n");
      gets(line);
      sscanf(line,"%e",&G);

    } /* end while */

  printf("\n\nWhat is the propagation delay of the");
  printf(" network? (in decimal real)\n");
  gets(line);
  sscanf(line,"%e",&a);

  while (a < 0.0)
    {
      printf("\nThe propagation delay must be greater");
      printf(" than 0.0");
      printf("\n\nWhat is the propagation delay of the");
      printf(" network? (in decimal real)\n");
      gets(line);
      sscanf(line,"%e",&a);

    } /* end while */

  term1 = term2 = term3 = term4 = 0.0;
  num = denom = 0.0;

  term1 = G * (exp (-G * (1.0 + a)));
  term2 = 1.0 + a - (exp (-a * G));
  num = term1 * term2;
```

```
  if (a == 0.0)
    {
      printf("\nA propagation delay of zero causes a");
      printf(" division by zero error.");
      printf("\nThis is equivalent to a throughput equal");
      printf(" to infinity.");

    } /* end if */
  else
    {
      term3 = (1.0 + a) * (1.0 - exp (-a * G));
      term4 = a * exp (-G * (1.0 + a));
      denom = term3 + term4;

      s = num / denom;

      printf("\nAt an offered rate");
      printf(" (packets/transmission-time) of:  %f",G);
      printf("\nwith a propagation delay of:  %f",a);
      printf("\nthe throughput of the network is:  %f",s);

    } /* end else */

  printf("\n\n");

  return (1);

}  /* end SLOTTED 1-PERSISTENT CSMA with INFINITE USERS */


/***** SLOTTED P_PERSISTENT CSMA with INFINITE USERS ******/
/*                                                        */
/*  Reference:                                            */
/*    Takagi, Hideaki, and Leonard Kleinrock, Throughput  */
/*      Analysis of Persistent CSMA Systems, IEEE         */
/*      Transactions on Communications, Volume COM-33     */
/*      no. 7, July, 1985, p. 630.                        */
/*                                                        */
/**********************************************************/

i_s_p_persistent_csma()

{

double term1, term2, term3, term4, term5, term6, num;
double term7, term8, term9, denom, last, sum, diff;
double a = 0.0, G = 0.0, p = 0.0, s = 0.0;
int k;


  printf("\n\nWhat is the propagation delay?\n");
  gets(line);
  sscanf(line,"%f",&a);
```

```
while (a < 0.0)
  {
    printf("\n\nThe propagation delay must be greater");
    printf(" than 0.0");
    printf("\nWhat is the propagation delay?\n");
    gets(line);
    sscanf(line,"%f",&a);
  }

printf("\n\nWhat is the offered traffic rate?\n");
gets(line);
sscanf(line,"%f",&G);

while (G < 0.0)
  {
    printf("\n\nThe offered traffic rate must be");
    printf(" greater than 0.0");
    printf("\nWhat is the offered traffic rate?\n");
    gets(line);
    sscanf(line,"%f",&G);
  }

printf("\n\nWhat is the level of persistence?\n");
gets(line);
sscanf(line,"%f",&p);

while ((p <= 0.0) || (p > 1.0))
  {
    printf("\n\nThe persistence level must be");
    printf(" between 0.0 and 1.0");
    printf("\nWhat is the level of persistence?\n");
    gets(line);
    sscanf(line,"%f",&p);
  }

printf("\n\nThis will take a minute, please wait...");


num = 0.0;
last = 0.0;
k = 0;

printf("\n\nCalculating numerator in formula");
printf("\n");

do
 {

  last = num;

  term1 = (1.0 - (pow((1.0 - p),(k + 2.0)))) / p;
  term2 = -1.0 * (k + 1.0);
  term3 = G * (pow((1.0 - p),(k + 1.0)));
  term4 = 1.0 * exp(term3 + ((a*G) * (term1 + term2)));
```

```
    term5 = p * (pow((1.0 - p),(1.0 * k)));
    term6 = a * (1.0 - (pow((1.0 - p),(k + 1.0))));
    num = (term5 + term6) * term4;

    ++k;
    num = num + last;
    diff = num - last;

    } while (((diff > 0.01) && (k <= ITERATIONS))
               || (num == diff) );

  num = G * num;
  printf("\nCalculating denominator in formula");
  printf("\n");

  sum = 0.0;
  last = 0.0;
  k = 1;

  do
   {

    last = sum;

    term7 = ((1.0 - (pow((1.0 - p),(k + 1.0)))) / p) - k;
    term8 = a * G * term7;
    sum = 1.0 * exp((G * pow((1.0 - p),(1.0*k))) + term8);

    ++k;
    sum = sum + last;
    diff = sum - last;

    } while ((diff > 0.01) && (k <= ITERATIONS));

  denom = (sum * a) + ((1 + a) * exp(G * (1.0 + a)));

  s = num / denom;

  printf("\n\nWith a propagation delay of %f,",a);
  printf("\nan offered traffic rate of %f,",G);
  printf("\nand a persistence level of %f,",p);
  printf("\nthe throughput is %f",s);
  printf("\n\nThe calculations for the numerator and");
  printf(" denominator are accurate to 0.01");
  printf("\n\n");

  return(1);

} /* end SLOTTED P-PERSISTENT CSMA INFINITE USERS */


/********* 1-PERSISTENT CSMA/CD with FINITE USERS *********/
/*                                                         */
/*  Reference:                                             */
```

```
/*    Takagi, Hideaki, and Leonard Kleinrock, Throughput   */
/*      Analysis of Persistent CSMA Systems, IEEE          */
/*      Transactions on Communications, Volume COM-33      */
/*      no. 7, July, 1985, p. 636.                         */
/*                                                         */
/***********************************************************/

csma_cd_1_persistent()
{

double a = 0.0, G = 0.0, p = 0.0, s = 0.0, b2 = 0.0;
double term1, term2, term3, term4, term5, num;
double term6, term7, term8, term9, denom;
int b;

  printf("\n\nWhat is the propagation delay?\n");
  gets(line);
  sscanf(line,"%f",&a);

  while (a < 0.0)
    {
      printf("\n\nThe propagation delay must be greater");
      printf(" than 0.0");
      printf("\nWhat is the propagation delay?\n");
      gets(line);
      sscanf(line,"%f",&a);
    }

  printf("\n\nWhat is the offered traffic rate?");
  printf(" (in decimal real)\n");
  gets(line);
  sscanf(line,"%e",&G);

  while (G < 0.0)
    {
      printf("\nThe offered traffic rate must be greater");
      printf(" than 0.0");
      printf("\nWhat is the offered traffic rate?");
      printf(" (in decimal real)\n");
      gets(line);
      sscanf(line,"%e",&G);

    } /* end while */

  printf("\n\nWhat is the collision detection factor?\n");
  gets(line);
  sscanf(line,"%f",&b2);

  while (b2 < 0.0)
    {
      printf("\n\nThe collision detection factor must be");
      printf(" greater than 0.0");
      printf("\n\nWhat is the collision detection");
      printf(" factor?\n");
```

```
      gets(line);
      sscanf(line,"%f",&b2);
    }

  b = b2 / a;

  term1 = (b2 * G / 2) + 1.0;
  term2 = (1.0 - exp(-2 * a * G));
  term3 = a * G * (exp(-2.0 * a * G)) / 2.0;
  term4 = G * exp(-G * (b2 + a));
  term5 = G * (1.0 + G) * exp(-G * (1.0 + (2.0 * a)));
  num = (term1 * term2 - term3) * term4 + term5;

  term6 = 1.0 * exp(-G * (1.0 + a));
  term7 = G * exp(-a * G);
  term8 = (1.0 - exp(-a * G)) * (1.0 + (G * (b2 + a)));
  term9 = 1.0 * exp(-b2 * G) * (1.0 - exp(-2.0*a*G)) / 2.0;
  denom = term6 + term7 + term8 + term9;

  if (denom >= 0.0)
    s = num / denom;
  else
    {
      s = 0.0;
      printf("\nDivision by zero error occurred.");
    }

  printf("\n\nWith a propagation delay of %f,",a);
  printf("\nan offered traffic rate of %f,",G);
  printf("\nand a collision detection factor of %f,",b2);
  printf("\n(or number of packets to detect a collision");
  printf(" equal to %d)",b);
  printf("\nthe capacity of the network is %f",s);

  return (1);

} /*  1-PERSISTENT CSMA/CD with INFINITE USERS */


/******* SLOTTED 1-PERSISTENT CSMA with FINITE USERS ******/
/*                                                        */
/*  Reference:                                            */
/*    Takagi, Hideaki, and Leonard Kleinrock, Throughput  */
/*      Analysis of Persistent CSMA Systems, IEEE         */
/*      Transactions on Communications, Volume COM-33     */
/*      no. 7, July, 1985, p. 630.                        */
/*                                                        */
/**********************************************************/


s_1_persistent_csma()
{

double term1, term2, term3, term4, term5, term6, num, denom;
```

```
double a = 0.0, g = 0.0, s = 0.0;
int m = 0;


  printf("\n\nWhat is the propagation delay of the");
  printf(" network?\n");
  gets(line);
  sscanf(line,"%f",&a);

  while (a < 0.0)
    {
      printf("\nThe propagation delay must be greater");
      printf(" than 0.0");
      printf("\nWhat is the propagation delay of the");
      printf(" network?\n");
      gets(line);
      sscanf(line,"%f",&a);
    }

  printf("\nWhat is the geometrical arrival rate");
  printf(" of a packet?\n");
  gets(line);
  sscanf(line,"%f",&g);

  while ((g < 0.0) || (g > 1.0))
    {
      printf("\nWhat is the geometrical arrival rate");
      printf(" of a packet?\n");
      gets(line);
      sscanf(line,"%f",&g);
    }

  printf("\nHow many workstations (users) are");
  printf(" on the network?\n");
  gets(line);
  sscanf(line,"%d",&m);

  while (m <= 0)
    {
      printf("\nHow many workstations (users) are");
      printf(" on the network?\n");
      gets(line);
      sscanf(line,"%d",&m);
    }

  term1 = term2 = term3 = term4 = term5 = term6 = 0.0;
  num = denom = 0.0;

  term1 = m * 1.0 * (pow((1.0 - g),
           ((1.0 * (m - 1)) * (1.0 + (1.0/a)))));
  term2 = 1 - (pow((1.0 - g),(1.0 + (1.0/a))));
  term3 = 1 - (pow((1.0 - g),(1.0 * m)));
  term4 = g * (pow((1.0 - g),((1.0 * m) + (1.0/a))));
  num = term1 * ((term2 * term3) + term4);
```

```
  term5 = (1.0 + a) * (1.0 - (pow((1.0 - g),(1.0 * m))));
  term6 = a * (pow((1.0 - g),((1.0 * m)*(1.0 + (1.0/a)))));
  denom = term5 + term6;

  s = num / denom;

  printf("\n\nWith a propagation delay of %f,",a);
  printf("\na geometric arrival rate of %f,",g);
  printf("\nand %d users : the throughput is %.16f",m,s);
  printf("\n\n");

  return (1);

}  /* end SLOTTED 1-PERSISTENT CSMA with FINITE USERS */


/****** SLOTTED P-PERSISTENT CSMA with FINITE USERS *******/
/*                                                        */
/*  Reference:                                            */
/*    Takagi, Hideaki, and Leonard Kleinrock, Throughput  */
/*      Analysis of Persistent CSMA Systems, IEEE         */
/*      Transactions on Communications, Volume COM-33     */
/*      no. 7, July, 1985, p. 630.                        */
/*                                                        */
/**********************************************************/


s_p_persistent_csma()
{

double a = 0.0, g = 0.0, p = 0.0, s = 0.0;
double term1, term2, term3;
double term1a, term1b, term1c, term1d, term1e;
double term2a, term2b, term2c, term2d, term2e;
double term3a, term3b, term3c, term3d, term3e;
double num, denom, last, diff;
int    m = 0, k = 0;

term1   = term2 = term3 = 0.0;
term1a  = term1b = term1c = term1d = term1e = 0.0;
term2a  = term2b = term2c = term2d = term2e = 0.0;
term3a  = term3b = term3c = term3d = term3e = 0.0;
num = denom = last = diff = 0.0;

   printf("\nWhat is the propagation delay of the");
   printf(" network?\n");
   gets(line);
   sscanf(line,"%f",&a);

   while (a < 0.0)
      {
        printf("\n\nThe propagation delay must be");
        printf(" between 0.0 and 1.0");
        printf("\nWhat is the propagation delay of the");
```

```
    printf(" network?\n");
    gets(line);
    sscanf(line,"%f",&a);
  }

printf("\nWhat is the geometric arrival rate?\n");
gets(line);
sscanf(line,"%f",&g);

while ((g < 0.0) || (g > 1.0))
  {
    printf("\n\nThe geometric arrival rate must be");
    printf(" between 0.0 and 1.0");
    printf("\nWhat is the geometric arrival rate?\n");
    gets(line);
    sscanf(line,"%f",&g);
  }

printf("\nHow many workstations (users) are on the");
printf(" network?\n");
gets(line);
sscanf(line,"%d",&m);

while (m <= 0)
  {
    printf("\nThe number of users must be greater");
    printf(" than 0");
    printf("\nHow many workstations (users) are on the");
    printf(" network?\n");
    gets(line);
    sscanf(line,"%d",&m);
  }

printf("\nWhat is the persistence level?\n");
gets(line);
sscanf(line,"%f",&p);

while ((p <= 0.0) || (p > 1.0))
  {
    printf("\nThe persistence level must be");
    printf(" between 0.0 and 1.0");
    printf("\nWhat is the persistence level?\n");
    gets(line);
    sscanf(line,"%f",&p);
  }

printf("\nCalculating...This could take a minute");
printf("...Please wait");
printf("\n");

k = 0;

if (p != g)
  {
```

```
do
  {
    last = num;

    term1a = 1.0 * pow((1.0 - p),(1.0 * k));
    term1b = 1.0 * pow((1.0 - g),(1.0 + (1.0/a)));
    term1c = p * (pow((1.0 - p),(1.0 * k)));
    term1d = g * (pow((1.0 - g),(1.0 * k)));
    term1e = p - g;

    term1 = term1a-(term1b*((term1c-term1d)/term1e));

    term2a = 1.0 * pow((1.0 - p),(1.0 * (k + 1)));
    term2b = p * (pow((1.0 - g),(1.0 + (1.0/a))));
    term2c = 1.0 * pow((1.0 - p),(1.0 * (k + 1)));
    term2d = 1.0 * pow((1.0 - g),(1.0 * (k + 1)));
    term2e = p - g;

    term2 = 1.0 * pow((term2a - (term2b * ((term2c -
                term2d) / term2e))),(1.0 * (m - 1)));

    num = term1 * term2;
    num = num + last;
    diff = num - last;
    ++k;

  } while (diff > 0.000001);

  num = p * m * num;

  k = 1;
  last = 0.0;

  do
    {
      last = denom;

      term3a = 1.0 * pow((1.0 - p),(1.0 * k));
      term3b = p * (pow((1.0 - g),(1.0 + (1.0/a))));
      term3c = 1.0 * pow((1.0 - p),(1.0 * k));
      term3d = 1.0 * pow((1.0 - g),(1.0 * k));
      term3e = p - g;

      term3 = 1.0 * pow((term3a - (term3b * ((term3c -
                  term3d) / term3e))),(1.0 * m));

      denom = term3;
      denom = denom + last;
      diff = denom - last;
      ++k;

    } while (diff > 0.000001);

denom = 1 + a + (a * denom);
```

```
    s = num / denom;


    printf("\nWith a propagation delay of %f,",a);
    printf("\na geometric arrival rate of %f,",g);
    printf("\na persistence level of %f, and %d users",p,m);
    printf("\nthe throughput is %f",s);
    printf("\n\nThe calculations for numerator and");
    printf(" denominator are accurate to 0.000001\n\n");

    } /* end if */
  else
    {
      printf("\nIn the formula implemented if the");
      printf(" persistent level is equal to");
      printf("\nthe geometric arrival rate, then a");
      printf(" division by zero error occurs.");

    } /* end else */

  return (1);

}  /* end SLOTTED P-PERSISTENT CSMA with FINITE USERS */


/************************************************************/
/******************* MAIN PROGRAM *************************/
/************************************************************/


main()

{

char response, selection, line[LINELENGTH];
int  i, done;

  do
    {
      done = 0;

      printf("\n");
      printf("\n*************  Network Capacity Formulas");
      printf(" Available:  **************");
      printf("\n*****                                       ");
      printf("                              *****");
      printf("\n");
      printf("\n    Infinite Users:");
      printf("\n        (A) Pure ALOHA (infinite users)");
      printf("\n        (B) Slotted ALOHA (infinite users)");
      printf("\n        (C) Nonpersistent CSMA (infinite");
      printf(" users)");
```

```
printf("\n        (D) Slotted Nonpersistent CSMA");
printf(" (infinite users)");
printf("\n");
printf("\n        (E) 1-Persistent CSMA (infinite");
printf(" users)");
printf("\n        (F) Slotted 1-Persistent CSMA");
printf(" (infinite users)");
printf("\n        (G) Slotted P-Persistent CSMA");
printf(" (infinite users)");
printf("\n        (H) 1-Persistent CSMA/CD");
printf(" (infinite users)");
printf("\n");
printf("\n    Finite Users:");
printf("\n        (I) Slotted 1-Persistent CSMA");
printf(" (finite users)");
printf("\n        (J) Slotted P-Persistent CSMA");
printf(" (finite users)");
printf("\n");
printf("\n    (Q) Quit");
printf("\n");
printf("\n*****************************************");
printf("***************************");
printf("\n");

do
  {
    printf("\nSelect the appropriate letter for");
    printf(" your choice: ");
    gets(line);
    sscanf(line,"%c",&selection);
  }
while (isalpha(selection) == 0);

switch (selection)
  {
    case 'a':
      i = i_pure_aloha();
      break;

    case 'A':
      i = i_pure_aloha();
      break;

    case 'b':
      i = i_slotted_aloha();
      break;

    case 'B':
      i = i_slotted_aloha();
      break;

    case 'c':
      i = i_nonpersistent();
      break;
```

```
case 'C':
  i = i_nonpersistent();
  break;

case 'd':
  i = i_s_nonpersistent();
  break;

case 'D':
  i = i_s_nonpersistent();
  break;

case 'e':
  i = i_1_persistent_csma();
  break;

case 'E':
  i = i_1_persistent_csma();
  break;

case 'f':
  i = i_s_1_persistent_csma();
  break;

case 'F':
  i = i_s_1_persistent_csma();
  break;

case 'g':
  i = i_s_p_persistent_csma();
  break;

case 'G':
  i = i_s_p_persistent_csma();
  break;

case 'h':
  i = csma_cd_1_persistent();
  break;

case 'H':
  i = csma_cd_1_persistent();
  break;

case 'i':
  i = s_1_persistent_csma();
  break;

case 'I':
  i = s_1_persistent_csma();
  break;

case 'j':
  i = s_p_persistent_csma();
```

```
          break;

        case 'J':
          i = s_p_persistent_csma();
          break;

        case 'q':
          done = 1;
          break;

        case 'Q':
          done = 1;
          break;

        default:
          printf("\nInvalid selection.\n");

      } /* end switch */

    if (done == 0)
      {
        printf("\n\nRun program again (y/n)? ");
        gets(line);
        sscanf(line,"%c",&response);

      } /* end if */

  } /* do-while */

  while ( (done == 0) &&
          ( (response == 'y') || (response == 'Y') ) );

  printf("\n\n");

} /* end main */
```

Appendix J

```
/*************************************************************/
/*                                                           */
/*  Program:  Implementation of Throughput and Delay-        */
/*            Time Formulas for Networks                     */
/*                                                           */
/*  Author:   Monte L. Hall                                  */
/*                                                           */
/*  Purpose:  Masters Project Implementation                 */
/*                                                           */
/*  This program will calculate the throughput of a net-     */
/*  work and the time required to send a certain number      */
/*  of packets, given the following input parameters:        */
/*                                                           */
/*      (1)  Capacity of the Network                         */
/*      (2)  Baud Rate (Speed) of the Network                */
/*      (3)  Number of Bits in One Packet                    */
/*      (4)  Number of Packets to Transmit                   */
/*                                                           */
/*************************************************************/

#include <stdio.h>

#include <ctype.h>

#include <math.h>

#define  LINELENGTH  80


main ()

{

double capacity, bit_capacity, thruput, delay, msec;
int    speed, bits_in_packet, number_packets, total_bits;
char line[LINELENGTH];

  printf("\n**********************************************");
  printf("******************");
  printf("\n\n");
  printf("\nWhat is the capacity of the network ");
  printf(" (decimal)?  ");
  gets(line);
  sscanf(line,"%e",&capacity);

  while ((capacity <= 0.0) || (capacity > 1.0))
    {
      printf("\nThe capacity of the network must be a ");
      printf("decimal number > 0.0");
```

```
    printf("\nWhat is the capacity of the network ");
    printf(" (decimal)?  ");
    gets(line);
    sscanf(line,"%e",&capacity);

  } /* end while */

printf("\nWhat is the baud rate (speed) of the network");
printf("  (integer bps)?  ");
gets(line);
sscanf(line,"%d",&speed);

while (speed <= 0)
  {
    printf("\nThe speed of the network must be an ");
    printf("integer greater than zero.");
    printf("\n\nWhat is the baud rate (speed) of the ");
    printf("network  (integer bps)?  ");
    gets(line);
    sscanf(line,"%d",&speed);

  } /* end while */

printf("\nHow many bits are in one packet (integer)?  ");
gets(line);
sscanf(line,"%d",&bits_in_packet);

while (bits_in_packet <= 0)
  {
    printf("\nThe number of bits in one packet must ");
    printf("be an integer greater than zero.");
    printf("\n\nHow many bits are in one ");
    printf("packet (integer)?  ");
    gets(line);
    sscanf(line,"%d",&bits_in_packet);

  } /* end while */

printf("\nHow many packets are in need of ");
printf("transmission  (integer)?  ");
gets(line);
sscanf(line,"%d",&number_packets);

while (number_packets <= 0)
  {
    printf("\nThe number of packets to transmit must ");
    printf("be an integer greater than zero.");
    printf("\n\nHow many packets are in need of ");
    printf("transmission  (integer)?  ");
    gets(line);
    sscanf(line,"%d",&number_packets);

  } /* end while */
```

```
  bit_capacity = 1.0 * capacity * speed;
  thruput = bit_capacity / bits_in_packet;


  total_bits = bits_in_packet * number_packets;
  delay = total_bits / bit_capacity;
  msec = delay * 1000;


  printf("\n\n*****************************************");
  printf("**********************");
  printf("\n");
  printf("\nThe throughput of the network is %f ",thruput);
  printf(" packets per second.",thruput);
  printf("\n\nThe time required to send");
  printf(" %d",number_packets);
  printf(" packets is %f",delay);
  printf("\nor %f milliseconds (ms).",msec);
  printf("\n");
  printf("\n******************************************");
  printf("*********************");
  printf("\n\n\n");

} /* end main */
```

SIMULATING A DISTRIBUTED FILE SYSTEM WITH VARIOUS TYPES OF NETWORKS

by

MONTE L. HALL

B. S., Kansas State University, 1986

AN ABSTRACT OF A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1988

Abstract

A distributed file system with dynamic locking of files is modeled with a queueing network taken from [Hac 1986]. The model contains a network link connecting host computers. Formulas for throughput of various networks, along with packet sizes and lengths, help determine a service time for this network link. This service time represents the delay encountered in transferring packets over a certain type of network. Various performance statistics are derived for both a one-host and a two-host model with and without files being shared. A discussion of the effect of the network on this model is also given.