Discrete-Event Simulation

A. Udaya Shankar

Department of Computer Science
University of Maryland
College Park, Maryland 20742

January, 1991

1. Introduction

Discrete-event simulation is a powerful computing technique for understanding the behavior of systems. By a *system*, we mean a collection of entities (e.g., people and machines) that interact over time. The particular nature of the system and the properties we wish to understand can vary. Here are three examples:

- A natural scientist may be interested in a system of wolves and sheep, where the number of wolves changes with a constant birth rate and a death rate that is inversely proportional to the number of sheep, and the number of sheep changes with a constant birth rate and a death rate that is directly proportional to the number of wolves. The scientist would like to know the following: Do the number of wolves and the number of sheep stabilize in the long run, and if so to what values? Or do they vary cyclically, and if so with what period and phase?
- A computer scientist may be interested in a system of jobs that circulate in a network of servers (e.g., CPU's and I/O devices). The computer scientist would like to know whether a particular server is a "bottleneck", i.e., in the long run, is that server always busy while the other servers are mostly idle.
- A classical system example is a queueing system with a single server. Here, customers arrive with certain service requirements, get served in some order, say first-come-first-served, and depart when their service is completed. Note that a customer who arrives when the server is busy has to wait (in a queue). For this system, we would like to determine the average waiting time for customers, the average number of customers in the system, the fraction of time the server is busy, etc.

In general, to determine whether a system satisfies a property, we have to come up with a mathematical model of the system. In discrete-event simulation, the models are restricted to so-called *discrete-event models*. Here, a set of *system states* is specified for the system, and the evolution of the system is viewed as a sequence of the form:

$$< s_0, (e_0, t_0), s_1, (e_1, t_1), s_2, \cdots >$$

where the s_i's are system states, the e_i's are system events, and the t_i's are nonnegative numbers representing event occurrence times. Informally, the above sequence means that the system started, say at time 0, in state s_0; then event e_0 occurred at time t_0 taking the system to state s_1; then event e_1 occurred at time t_1 taking the system to state s_2; and so on. Each event occurrence is assumed to take zero time. The t_i's are required to be nondecreasing, i.e., $t_i \le t_{i+1}$ for every i. (We cannot insist that $t_i < t_{i+1}$ because it is the case in discrete-event models that two unrelated events can occur at the same time. However, in the discrete-event models that we shall consider, there are at most a finite number of transitions over any finite time interval.)

Given the evolution of a system, we can determine its properties (e.g., does it reach steady state, is it cyclic, etc.) and evaluate appropriate *performance measures* (e.g., the steady state values, the cycle period, etc.). Thus, our objective is an efficient method to generate evolutions and evaluate properties and performance measures.

In general, there is a set of system parameters, referred to as *input parameters*, that determines the evolution of the system, and hence the properties and performance measures. For example, the input parameters to the queueing system are the customer service requirements and arrival times. Typically, we want to describe the input parameters of a system *stochastically* (or probabilistically), instead of deterministically. That is, instead of fixing the input parameter values deterministically, we let them be *random variables*, taking values from some domain with some probability distribution. Each set of input parameter values gives rise to a unique evolution. The objective is to obtain performance measures averaged over all such evolutions.

There are two reasons for introducing random variables. First, for most real-life system, we do not have exact characterizations of the input parameters. Hence, using probabilistic inputs makes the results of the analysis more robust. Second, even if we do have an exact characterization of the input parameters, it is often computationally too expensively or analytically intractable to take them into account.

Organization of the notes

In Section 2, we define some performance measures for the single-server queueing system. In Section 3, we describe the general structure of event-driven simulators. In Section 4, we describe a deterministic simulator for the single-server queue. In Section 5, we describe a stochastic simulator for the single-server queue. In Section 6, we describe how to generate random variables of given distributions. In Section 7, we describe a simulation project.

2. Performance Measures for Queueing Systems

Consider the queueing system with a single server mentioned in Section 1. Let customer n denote the nth customer to arrive at the queueing system, for $n = 1, 2, \cdots$.

Let us represent the state of the system by the queue of customers in the system, in the order of their arrival. For example, < 3, 4, 5 > means that customers 3, 4 and 5 are in the system. By convention, the head of the queue is at the left. If the queue is not empty, then the customer at the head is being served. We use <> to denote an empty queue.

Let the events of the system be *Arrival*(n) denoting the arrival of customer n, and *Departure*(n) denoting the departure of customer n. (This assumes that if a customer completes service when other customers are waiting, then the next customer's service is started immediately; otherwise, we would need another event representing the start of service.)

Let S_n denote the service time of customer n; i.e., customer n requires the server's attention for S_n seconds. (Without loss of generality, we assume that time units are seconds.) Let TA_n denote the arrival time of customer n.

The following sequence represents an evolution of the system, assuming that S_n equals 2.0 seconds for all n, and TA_n equals $2.5n - 2.5$ seconds for odd n and $2.5n - 4$ seconds for even n (i.e., customers arrive at times $0.0, 1.0, 5.0, 6.0, 10.0, 11.0, \cdots$). For readability , each element of the evolution is listed on a new line. (Observe that the system evolution is cyclic with a period of 5 seconds.)

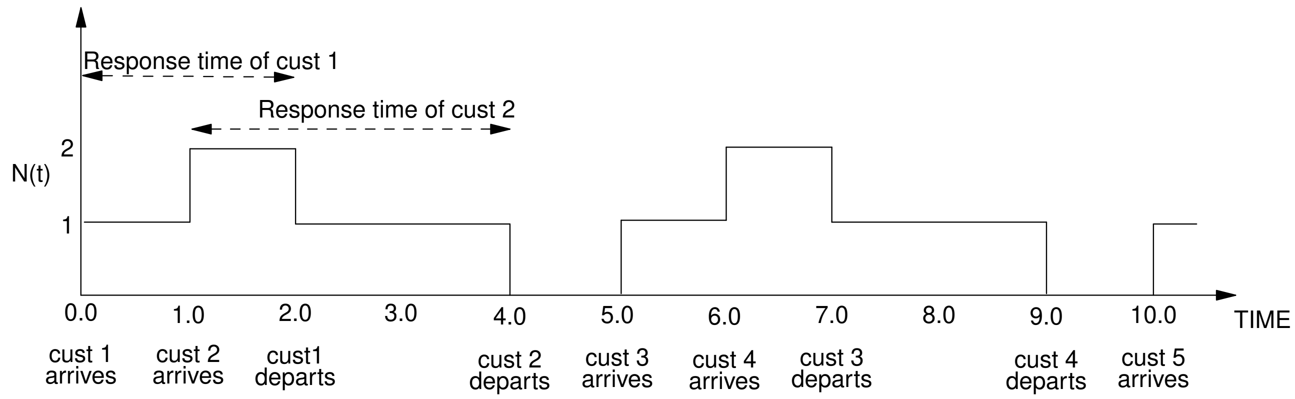
States	Event	Occurrence time
< >		
	Arrival(1)	0.0
<1>		
	Arrival(2)	1.0
<1, 2>		
	Departure(1)	2.0
<2>		
	Departure(2)	4.0
< >		
	Arrival(3)	5.0
<3>		
	Arrival(4)	6.0
<3, 4>		
	Departure(3)	7.0
<4>		

•
•
•

A queueing system has many *performance measures* of interest. We will look at some of them, namely, (1) the average number of customers (also called average system size), (2) the average response time, (3) the average waiting time, and (4) the throughput.

Average Number of Customers

Let $N(t)$ denote the number of customers in the system at time t. $N(t)$ is an integer-valued discontinuous function that increases by 1 at each arrival and decreases by 1 at each departure. The following graph shows $N(t)$ versus t.



For a given evolution, the *average number of customers in the system*, which we shall denote by $\bar{N}$, is defined to be the average of $N(t)$ over time for the evolution. Formally, if the time duration of the evolution is T seconds, then

$$\bar{N} = \frac{1}{T} \int_0^T N(t)dt$$

To illustrate, let us consider the evolution of our queueing system until just after the departure of customer 2. For this evolution, $\bar{N}$ equals 1.25. (It is obtained as follows. Customer 2 departs at time 4. $\int_0^4 N(t)dt$ (which is the area under $N(t)$ from time 0 to 4) equals 5. Thus, the average system size is $\frac{5}{4}$, which equals 1.25.)

In general, we want the "steady-state" value of $\bar{N}$, i.e., $\bar{N}$ for extremely "long" evolutions. Formally, we want

$$\bar{N} = \lim_{T \to \infty} \frac{1}{T} \int_0^T N(t)dt.$$

In the above queueing example, the steady-state $\bar{N}$ equals 1.0. (We can obtain it easily by noting that the evolution repeats itself every 5 seconds. Thus, it is sufficient to obtain $\bar{N}$ for any contiguous 5 second interval, such as [0, 5]. $\int_0^5 N(t)dt$ equals 5. Thus, the average system size is $\frac{5}{5}$, which equals 1.0.)

Average Response Time

The response time of customer n, denoted by R_n, is the time spent by the customer in the system. For a given evolution, the *average response time*, which we shall denote by $\bar{R}$, is the average of the R_n's for the customers departing in the evolution. Formally, if K customers depart in the evolution, then

$$\frac{1}{K} \sum_{i=1}^{K} R_i$$

To illustrate, let us consider the evolution of our queueing system until just after the departure of customer 2. For this evolution, $\bar{R}$ equals 2.5. (It is obtained as follows. There are two departures in this simulation, namely customers 1 and 2. The response time of customer 1 is 2.0 seconds. The response time of customer 2 is 3.0 seconds. Thus, the average response time is $\frac{2.0 + 3.0}{2}$, which equals 2.5.)

In general, we want the "steady-state" $\bar{R}$, i.e., for extremely "long" evolutions. That is, we want

$$\bar{R} = \lim_{K \to \infty} \frac{1}{K} \sum_{i=1}^{K} R_i$$

For the above queueing example, R_n equals 2.0 seconds for odd n and 3.0 seconds for even n. Thus, the steady-state $\bar{R}$ equals 2.5 seconds per customer.

Average Waiting Time

The waiting time of customer n, denoted by W_n, is defined by $W_n = R_n - S_n$. For an evolution, the average waiting time, denoted $\bar{W}$, is the average of the W_n's for the customers departing in the evolution. (For the above evolution, the steady-state $\bar{W}$ equals 0.5 seconds per customer.)

Throughput

For an evolution, the throughput, denoted by X, indicates the number of departures over the total time of the evolution. (For the above evolution, the steady-state X equals 0.4 customers per second.)

General comments

Note that $\bar{R}$ and $\bar{W}$ are customer averages, whereas $\bar{N}$ and X are time averages. In general, when we refer to a performance measure we mean its steady-state value, unless otherwise mentioned. Note that the steady-state averages do not always exist. For example in the above queueing system, if S_n were greater than 2.5, then $\bar{R}$, $\bar{W}$, and $\bar{N}$ would not exist.

The input parameters of the above queueing system are $\{S_n\}$ and $\{TA_n\}$. In the above description, we have described them deterministically. As mentioned in Section 1, we typically want to describe them probabilistically. For example, instead of having S_n equal 2.0 seconds for all n, we may want S_n to be a value between 1.7 to 2.3 seconds, such that each value in the range is chosen with uniform probability and successive values of S_n are chosen independently. We will see how to do this in Sections 5 and 6.

Observe that the above values for $\bar{N}$, $\bar{R}$ and X satisfy the following:

$$\bar{N} = \bar{R} \times X$$

This is not a coincidence. In fact, this is a very important relationship, called *Little's Law*. It holds for *any* general system in steady-state!

3. Event-Driven Simulators

In this section, we describe a classical method to generate evolutions and evaluate performance measures. The method does not store the evolution it generates; it only stores sufficient information to evaluate the desired performance measures. It proceeds in iterations. In each iteration, one event occurrence is simulated.

The following variables are used in the simulator:

Simulation Clock
: Nonnegative real number. At the start of each iteration, it indicates the time of the last event occurrence that has been simulated. Initially 0.

Event List
: A sequence of tuples of the form (e, t), where e is an event and t is a nonnegative real number not less than *Simulation Clock*. Each (e, t) tuple represents an event occurrence that is yet to be simulated. (Typically,

the tuples in the list are sorted according to t.) Initially, the list contains one or more event occurrence pairs to trigger the simulation (e.g. a customer arrival at time 0).

State Variables

At the start of each iteration, these variables indicate the state of the system after the last simulated event occurrence. Initially indicates an initial system state.

Performance Indicators

At the start of each iteration, these variables contain sufficient information to (1) evaluate the performance measures for the evolution that has been simulated, and (2) evaluate new values of the performance indicators if the evolution is extended by an event occurrence. Initially set to values corresponding to an empty evolution.

We next define procedures used in the simulator. For each event e of the system, the simulator has a procedure referred to as the **event handler** of e, and denoted by *Routine*(e, t). Here, t is a parameter representing an occurrence time. *Routine*(e, t) specifies the effect on the system due to an occurrence of event e at time t. It can update the system state and performance indicators, cause events to occur in the future (referred to as *scheduling events*), and cause scheduled events to not occur (i.e. unschedule them). The statements of *Routine*(e, t) can do the following:

- Access (read and write) *State Variables* and *Performance Indicators*.
- Read *Simulation Clock* (the value may be needed to update the performance indicators).
- Make calls to the procedures *Schedule*(f, s) and *Remove*(f, s), defined as follows, where f is an event and s is a nonnegative real number strictly greater than t:

  Schedule(f, s) = Enter the tuple (f, s) into *Event List*.

  Remove(f, s) = Remove the tuple (f, s) from *Event List*.
  (*Remove*(f, s) assumes that (f, s) is in *Event List*.)

To complete the description of the simulator, we now define a procedure, referred to as *Simulate*, representing the ‘‘main program’’:

```
Simulate =
 while ‘‘simulation not over’’ do
 begin
     Pick up an (e, t) tuple with minimum t from Event List;
     Call Routine(e, t);
        (updates system state and performance indicators, and
         perhaps schedules new event occurrences and removes scheduled event occurrences)
     Simulation Clock ← t
 end
```

Many particular conditions can be used for the generic ‘‘simulation not over’’ in the above procedure. For example, *Simulation Clock* $\geq$ 10000 seconds, or *Number of events simulated* $\geq$ 1000. It may depend on the particular system being simulated. For example, in simulating a queueing system, we can use *Number of Departures* ≥ 10000. If we know that the queueing system eventually reaches steady state, we may use the condition *Standard Deviation of Throughput* $\leq 0.001 \times$ *Average Throughput*. (The variables *Number of events simulated*, *Number of Departures*, *Standard Deviation of Throughput*, and *Average Throughput*, would be performance indicators.)

Given a simulator of a system, we refer to the *State Variables*, *Performance Indicators*, and the event handlers as the **simulation model** of the system. Note that the simulation model is the only system-specific part of the simulator.

Stochastic simulation models

Above, we have described deterministic simulation models. Recall that we typically want to simulate stochastic models. To do this, when we schedule an event to occur at time t, we allow t to be a random variable with some distribution.

For a distribution F, let *Random*(F) denote a function that returns a random number distributed according to F. Successive calls to *Random*(F) return numbers that are statistically independent. We allow event handlers to contain *Schedule*(f, s) statements where s is an expression involving *Random*(F)'s, rather than just deterministic functions. (Later, we will describe how to obtain *Random*(F).)

4. A Deterministic Simulator for the Single-Server Queue

In this section, we obtain a deterministic simulation model of the single-server queueing system described in Section 2. This, together with the ''main program'' *Simulate*, forms a complete simulator.

Let us assume that S_n and TA_n are some arbitrary deterministic functions of n.[1] Let us also assume that the queueing system is initially empty. We have the following variables; the comments associated with the variables hold whenever control comes to the start of the while loop in the main program *Simulate*:

Q: queue of (*CustomerId*, *ArrivalTime*). Initially <>.
The queue of customers in the system in order of arrival, along with their arrival times.

NumDepartures: integer. Initially 0.
The number of departures in the system since beginning of simulation.

SystSize: integer. Initially 0.
The number of customers in the system.

AccumSystSize1: real. Initially 0.
Integral of *SystSize* with respect to time. Sufficient to obtain the mean *SystSize*.

AccumSystSize2: real. Initially 0.
Integral of $SystSize^2$ with respect to time. Sufficient to obtain the second moment of *SystSize*.

AccumResponseTime1: real. Initially 0.
Sum of the response times of all departed customers. Sufficient to obtain the mean response time.

AccumResponseTime2: real. Initially 0.
Sum of the squares of the response times of all departed customers. Sufficient to obtain the second moment of the response time.

This completes the description of the variables of the simulator. Q is the only state variable. All the other variables are performance indicators. Observe that the system state here is different from that in Section 2. There, we identified customer n by the integer n, whereas here we also add the arrival time TA_n. We need this information to compute the response time.

The set of events of the system is $\{Arrival(n)$: for $n = 1, 2, \cdots\} \cup \{Departure\}$. We next define the event handlers:

---

[1]For example in Section 2, S_n equals 2.0 seconds for all n, and TA_n equals $2.5n - 2.5$ seconds for odd n and $2.5n - 4$ seconds for even n.

$Routine(Arrival(n), t) =$
    Append (n, t) to the tail of Q;
    $Schedule(Arrival(n+1), TA_{n+1})$;
    if $SystSize = 0$ then $Schedule(Departure(n), t + S_n)$;
    $UpdateStatePerformanceIndicators$;
    $SystSize \leftarrow SystSize + 1$;
end procedure

$Routine(Departure, t) =$
    (Assumes Q is not empty)
    $u \leftarrow Head(Q).\ ArrivalTime$;
    Remove the element at the head of Q;
    if $SystSize \geq 2$
      then begin
            $m \leftarrow Head(Q).\ CustomerId$;
            $Schedule(Departure(m), t + S_m)$
            end;
    $UpdateStatePerformanceIndicators$;
    $NumDepartures \leftarrow NumDepartures + 1$;
    $SystSize \leftarrow SystSize - 1$;
    $AccumResponseTime1 \leftarrow AccumResponseTime1 + (t - u)$;
    $AccumResponseTime2 \leftarrow AccumResponseTime2 + (t - u)^2$;
end procedure

where

$UpdateStatePerformanceIndicators =$
    $AccumSystSize1 \leftarrow AccumSystSize1 + (t - Simulation\ Clock) \times SystSize$;
    $AccumSystSize2 \leftarrow AccumSystSize2 + (t - Simulation\ Clock) \times (SystSize^2)$;
end procedure

At the end of the simulation (i.e., when procedure *Simulate* has finished execution) or at any point during the simulation, the following performance measures can be computed:

$$AverageResponseTime = \frac{AccumResponseTime1}{NumDepartures}$$

$$Standard\ Deviation\ of\ Response\ Time = \left(\frac{AccumResponseTime2}{NumDepartures} - AverageResponseTime^2\right)^{\frac{1}{2}}$$

$$AverageSystemSize = \frac{AccumSystSize1}{Simulation\ Clock}$$

$$Standard\ Deviation\ of\ System\ Size = \left(\frac{AccumSystSize2}{Simulation\ Clock} - AverageSystemSize^2\right)^{\frac{1}{2}}$$

5. A Stochastic Simulator for the Single-Server Queue

We now obtain a stochastic simulation model of the single-server queueing system. Define the interarrival time $A_n = TA_n - TA_{n-1}$. (It turns out to be more convenient to describe interarrival times than arrival times.) Let the interarrival times $\{A_n\}$ to be random variables that are independent and identically distributed with the distribution F_A. Let the service times $\{S_n\}$ be random variables that are independent and identically distributed with the distribution F_S.

The simulation model has the following variables:

Q: queue of *ArrivalTime*. Initially <>.
  The queue of customers in the system in order of arrival. Each customer is identified only by its arrival time

(this turns out to be adequate).

NumDepartures, *SystSize*, *AccumSystSize1*, *AccumSystSize2*, *AccumResponseTime1*, and *AccumResponseTime2* are defined as in the deterministic model above.

For the stochastic model, it is sufficient to consider only two events: *Arrival* and *Departure*. The event handlers are as follows:

$Routine(Arrival, t) =$
  Append t to the tail of Q;
  $Schedule(Arrival, t + Random(F_A))$;
  if $SystSize = 0$ then $Schedule(Departure, t + Random(F_S))$;
  UpdateStatePerformanceIndicators;
  $SystSize \leftarrow SystSize + 1$;
end procedure

$Routine(Departure, t) =$
  (Assumes Q is not empty)
  $u \leftarrow Head(Q)$;
  Remove the element at the head of Q;
  if $SystSize \geq 2$ then $Schedule(Departure, t + Random(F_S))$;
  UpdateStatePerformanceIndicators;
  $NumDepartures \leftarrow NumDepartures + 1$;
  $SystSize \leftarrow SystSize - 1$;
  $AccumResponseTime1 \leftarrow AccumResponseTime1 + (t - u)$;
  $AccumResponseTime2 \leftarrow AccumResponseTime2 + (t - u)^2$;
end procedure

where *UpdateStatePerformanceIndicators* is defined as in the deterministic model.

An example execution

The above state variables and event handlers define the simulation model. This, together with the main program— i.e. the variables *Simulation Clock* and *Event List*, and the procedure *Simulate*— make up the simulator.

To help the reader understand how the simulator works, we trace the execution of the simulator for a few event occurrences. In the trace, we give the simulation state (i.e. the values of the variables) each time that control comes to statement *while simulation not over* in the main program.[2] Between successive states, we briefly describe the execution of the relevant event handler. While going through the trace, the reader may find it convenient to plot $N(t)$ versus t as the simulation proceeds.

Initial simulation state

Simulation Clock = 0.0
Event List = $< (Arrival, 0.0) >$
$Q = <>$
NumDepartures = 0
SystSize = 0
AccumSystSize1 = 0.0
AccumResponseTime1 = 0.0

Handle next event occurrence

Remove the next event occurrence from *Event List* and execute the appropriate event handler. In this case, the

[2]For brevity, we omit indicating the values of *AccumSystSize2* and *AccumResponseTime2*.

next event occurrence is $(Arrival, 0.0)$. From the body of $Routine(Arrival, 0.0)$, we see that it schedules a new arrival at time $0.0 + Random(F_A)$ and a departure at time $0.0 + Random(F_S)$ (the departure is because the system was empty prior to this arrival event). Let us assume that $Random(F_A)$ returned 2.1 and $Random(F_S)$ returned 1.8. The resulting simulation state is given below. (Note that the state would be different if $Random(F_A)$ had returned a smaller value than $Random(F_S)$.)

Resulting simulation state

$Simulation\ Clock = 0.0$
$Event\ List = <\ (Departure, 1.8)\ ,\ (Arrival, 2.1)\ >$
$Q = <\ (0.0)\ >$
$NumDepartures = 0$
$SystSize = 1$
$AccumSystSize1 = 0.0 + 0 \times 0.0 = 0.0$
$AccumResponseTime1 = 0.0$

Handle next event occurrence

The next event occurrence in *Event List* is $(Departure, 1.8)$. From the body of $Routine(Departure, 1.8)$, we see that no event is scheduled (because the system is empty after this departure event). The resulting simulation state is given below.

Resulting simulation state

$Simulation\ Clock = 1.8$
$Event\ List = <\ (Arrival, 2.1)\ >$
$Q = <>$
$NumDepartures = 1$
$SystSize = 0$
$AccumSystSize1 = 0.0 + 1 \times (1.8 - 0.0) = 1.8$
$AccumResponseTime1 = 0.0 + (1.8 - 0.0) = 1.8$

Handle next event occurrence

The next event occurrence is $(Arrival, 2.1)$. Its event handler schedules a new arrival at time $2.1 + Random(F_A)$ and a departure at time $2.1 + Random(F_S)$. Let us assume that $Random(F_A)$ returned 1.7 and $Random(F_S)$ returned 3.1.

Resulting simulation state

$Simulation\ Clock = 2.1$
$Event\ List = <\ (Arrival, 3.8)\ >\ ,\ (Departure, 5.2)\ >$
$Q = <\ (2.1)\ >$
$NumDepartures = 1$
$SystSize = 1$
$AccumSystSize1 = 1.8 + 0 \times (2.1 - 1.8) = 1.8$
$AccumResponseTime1 = 1.8$

Handle next event occurrence

The next event occurrence is $(Arrival, 3.8)$. Its event handler schedules a new arrival at time $3.8 + Random(F_A)$ (note that it does not schedule a departure because the server was busy prior to this arrival event). Let us assume that $Random(F_A)$ returned 1.1.

Resulting simulation state

Simulation Clock = 3.8
Event List = < (*Arrival*, 4.9) > , (*Departure*, 5.2) >
Q = < (2.1) , (3.8) >
NumDepartures = 1
SystSize = 2
AccumSystSize1 = $1.8 + 1 \times (3.8 - 2.1) = 3.5$
AccumResponseTime1 = 1.8

Handle next event occurrence

The next event occurrence is (*Arrival*, 4.9). Its event handler schedules a new arrival at time $4.9 + Random(F_A)$. Let us assume that $Random(F_A)$ returned 2.6.

Resulting simulation state

Simulation Clock = 4.9
Event List = < (*Departure*, 5.2) > , (*Arrival*, 7.5) >
Q = < (2.1) , (3.8) , (4.9) >
NumDepartures = 1
SystSize = 3
AccumSystSize1 = $3.5 + 2 \times (4.9 - 3.8) = 5.7$
AccumResponseTime1 = 1.8

Handle next event occurrence

The next event occurrence is (*Departure*, 5.2). It schedules a new departure at time $5.2 + Random(F_S)$. Let us assume that $Random(F_S)$ returned 1.1.

Resulting simulation state

Simulation Clock = 5.2
Event List = < (*Departure*, 6.3) > , (*Arrival*, 7.5) >
Q = < (3.8) , (4.9) >
NumDepartures = 2
SystSize = 2
AccumSystSize1 = $5.7 + 3 \times (5.2 - 4.9) = 6.6$
AccumResponseTime1 = $1.8 + (5.2 - 2.1) = 4.9$

Handle next event occurrence

The next event occurrence is (*Departure*, 6.3). It schedules a new departure at time $6.3 + Random(F_S)$. Let us assume that $Random(F_S)$ returned 1.5.

Resulting simulation state

Simulation Clock = 6.3
Event List = < (*Arrival*, 7.5) > , (*Departure*, 7.8) >
Q = < (4.9) >
NumDepartures = 3
SystSize = 1
AccumSystSize1 = $6.6 + 2 \times (6.3 - 5.2) = 8.8$
AccumResponseTime1 = $4.9 + (6.3 - 3.8) = 7.4$

If we stop the simulation at this point, then the *AverageSystemSize* would be $\frac{8.8}{6.3} = 1.39$ and the *AverageResponseTime* would be $\frac{7.4}{3} = 2.47$. Of course, typically we would continue the simulation for (at least) hundreds of departures.

6. Generating Random Numbers

Before describing how to generate the random numbers needed for your project, we will first give some background. If you do not understand the background, don't worry. Just make sure you understand the procedures given in the last part of this section.

Random variables are defined by their distribution functions. There are two kinds of distribution functions: *discrete* and *continuous*.

A discrete distribution function has a subset of the integers as its domain, e.g., $\{0, 1\}$, $\{0, 1, \cdots\}$. If a random variable R has a discrete distribution F, then it means the following:

$$\mathit{Probability}(R = n) = F(n) \qquad \text{for every } n \text{ in the domain of } F$$

A continuous distribution function has the real numbers as its domain. If a random variable R has a continuous distribution F, then it means the following:

$$\lim_{\delta \to 0} \mathit{Probability}(R \in (x,\ x + \delta]) = F(x)\delta \qquad \text{for every real } x$$

In this project, we will be concerned with one discrete distribution, namely the *Bernoulli*, and two continuous distributions, namely the *Uniform* and the *Exponential*. We now define them:

- The Bernoulli distribution has a real-valued parameter, say p, where $0 \le p \le 1$, and a domain of two values, say $\{0, 1\}$. A Bernoulli distribution F with parameter p is defined by

$$F(n) = \begin{cases} p & n = 0 \\ 1 - p & n = 1 \end{cases}$$

- The Uniform distribution has two real-valued parameters, say a and b, where $a < b$. A uniform distribution F with parameters a and b is defined by

$$F(x) = \begin{cases} \dfrac{1}{b - a} & x \in [a, b] \\ 0 & x \notin [a, b] \end{cases}$$

- The Exponential distribution has one real-valued parameter, say s, where $s > 0$. An exponential distribution F with parameter s is defined by

$$F(x) = \begin{cases} \dfrac{1}{s} \exp(-\dfrac{x}{s}) & x \ge 0 \\ 0 & x < 0 \end{cases}$$

Most computer systems have a function, say *Random*, that returns a random variable of uniform distribution with parameters 0, 1. Using *Random*, we can obtain random number generators for other distributions.

- The following procedure returns a random variable that has a Bernoulli distribution with parameter p:

```
Bernoulli(p) =
    if Random < p then return(1) else return(0)
end procedure
```

- The following procedure returns a random variable that has a Uniform distribution with parameters a and b:

```
Uniform(a, b) =
    return (a + (b − a) × Random)
end procedure
```

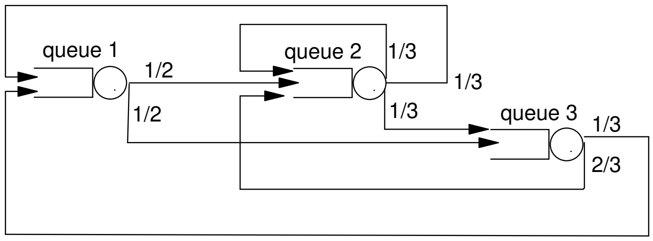
- The following procedure returns a random variable that has an Exponential distribution with parameter s (below, *Ln* stands for the natural logarithm):

Exponential(s) =
    return ($-\, s \times Ln(Random)$)
end procedure

7. Project

Your project involves the simulation of a closed network of (single-server) queues. By "closed network", we mean that (1) there are no external arrivals, and (2) when a customer leaves one queue, it joins another queue. Closed networks are very useful for modeling jobs in a multi-programming system, packets in a communication network, etc.

Your project will simulate a network with 3 queues, as shown in the following figure:



As indicated in the figure, a customer that departs from queue 1 joins queue 2 with probability 1/2 and joins queue 3 with probability 1/2. A customer that departs from queue 2 joins queue 3 with probability 1/3, joins queue 1 with probability 1/3, and returns to queue 2 with probability 1/3. A customer that departs from queue 3 joins queue 1 with probability 1/3 and joins queue 2 with probability 2/3. Each departure from a queue is routed according to a Bernoulli distribution. Successive departures from a queue are routed independently.

Queues 1 and 2 use FCFS (first-come-first-served) discipline, as in the single-server examples of Section 2. Queue 3 uses LCFS-PR (last-come-first-serve preemptive resume) discipline. Here, if a customer *A* arrives when another customer *B* is being served, customer *B* is preempted and put on a stack of preempted customers, and customer *A*'s service starts immediately. When the server completes service of a customer, it resumes service of the customer at the top of the stack. (**Make sure you understand the LCFS-PR discipline**.)

Develop a simulator for the above network, with input and output specifications as described below.

Input Specification

Your program should prompt for the following inputs, in the order given below. For each prompt, the user should type in one of the allowed inputs; below, we use "**or**" to separate different types of inputs for the same prompt. Begin each prompt on a new line. Each input ends with a new line.

- *Prompt*: Maximum Simulation time:
  Input: x

  x is the maximum time for which the simulation should be run. i.e., stop the simulation when *Simulation Clock* > x.

- *Prompt*: Number of customers.
  Input: n

  n is an positive integer indicating the number of customers in the system. Initially, let all the customers be in queue 1.

- *Prompt*: Service distribution at 1.
  Input: U a b **or** E s

The service distribution at server 1. The first input, U a b, indicates a uniform distribution with parameters a and b. The second input, E s, indicates an exponential distribution with parameter s. For each input, the first value is a character while the other(s) are reals.

- *Prompt*: Service distribution at 2.
  Input: U a b **or** E s
- *Prompt*: Service distribution at 3.
  Input: U a b **or** E s
- *Prompt*: Trace?
  Input: Y **or** N

  The input is a character, indicating whether the output trace feature is to be on (Y) or off (N).

Output Specification

Your program should output the following, in the order given below. Begin each output on a new line.

- *Output*: Maximum simulation time = x

  x equals the maximum simulation time.
- *Output*: Number of customers = x

  x equals the number of customers.
- *Output*: Average Service Time at queue 1 = x

  x equals the mean of the service times actually generated during your simulation. This should be close to, but not necessarily exactly the same as, the mean of the distribution that was input. For example, if server 1 was exponentially distributed with parameter s, then $x \approx s$. If server 1 was uniformly distributed with parameters a and b, then $x \approx \frac{a+b}{2}$.
- *Output*: Standard Deviation of the Service Time at queue 1 = x

  x equals the standard deviation of the service times actually generated during your simulation.
- *Output*: Average System Size at queue 1 = x

  x equals the mean number of customers in queue 1.
- *Output*: Standard Deviation of the System Size at queue 1 = x

  x equals the standard deviation of the number of customers in queue 1.
- *Output*: Average Response Time at queue 1 = x

  x equals the mean of the response times experienced by the customers in queue 1.
- *Output*: Standard Deviation of the Response Time at queue 1 = x

  x equals the standard deviation of the response times experienced by the customers in queue 1.
- *Output*: Throughput of queue 1 = x

  x equals the throughput of queue 1, i.e., number of customers who exited queue 1 divided by the total time.
- *Output*: Average Service Time at queue 2 = x

- *Output*: Standard Deviation of the Service Time at queue 2 = x
- *Output*: Average System Size at queue 2 = x
- *Output*: Standard Deviation of the System Size at queue 2 = x
- *Output*: Average Response Time at queue 2 = x
- *Output*: Standard Deviation of the Response Time at queue 2 = x
- *Output*: Throughput of queue 2 = x

- *Output*: Average Service Time at queue 3 = x
- *Output*: Standard Deviation of the Service Time at queue 3 = x
- *Output*: Average System Size at queue 3 = x
- *Output*: Standard Deviation of the System Size at queue 3 = x
- *Output*: Average Response Time at queue 3 = x
- *Output*: Standard Deviation of the Response Time at queue 3 = x
- *Output*: Throughput of queue 3 = x

- *Output*: Average System Cycle Time = x

  Time between successive departures of a given customer from queue 1, averaged over all customers.

- *Output*: Standard Deviation of System Cycle Time = x

  x equals the standard deviation of the time between successive departures of a given customer from queue 1, averaged over all customers.

- *Output*: Trace = sequence of (i, j, t) tuples.

  Each (i, j, t) tuple means that a customer left queue i and joined queue j at time t. The tuples in the sequence are in the order generated in the simulation. This output should be done if and only if the input Trace? flag was set to Y.