6. Discrete Event Simulation

As it should be obvious by now, models that are amenable to analysis, such as queuing models, ignore (or abstract out) many of the details often present in real systems. For that reason, we often rely on simulations. In a simulation, a more detailed model of the system is allowed to "execute" as a program and observations made can be used to make conclusions about the system being modeled.

6.1 Simulation Approaches

It makes sense to distinguish between two types of simulations: *continuous* simulation and *discrete*, or *event-based* simulation.

6.1.1 Continuous Simulation

There are many instances in which the state of the system changes continuously. In other words, it is not possible to attribute changes in the system to discrete instantaneous events. As an example, consider how one would model and simulate the "weather" (for weather forecasting purposes, let's say). Clearly, changes in temperature, wind, etc. are governed by continuous processes, typically modeled by differential equations. Other examples of continuous simulations are those used to evaluate electric properties of VLSI circuits. Given that the underlying processes are changing continuously, to simulate such systems we typically approximate the time continuum with a discrete periodic simulation by using "ticks," reevaluating the system every tick.

6.1.2 Discrete (Event) Simulation

A discrete event simulator assumes that the state of the system changes as a result of specific events and that it remains unchanged (static) between events. Furthermore, events are assumed to be instantaneous and thus consume 0 time. Clearly, discrete event simulations will be useful to simulate systems for which it could be safely assumed that the concept of an event exists. The operation of a clocked system (such as a CPU) is such an example, where the only time the system changes is when the clock fires. Another example, of course, is queuing systems. Here it is clear that the state of the queues in the system will not change unless an event (such as a birth or death) occurs in some queue.

In a computing system, it is natural to assume that changes in the state of the system will be due to specific events (e.g., arrival of a packet to a router, completion of service from an I/O device, etc.). Thus, for the purposes of the simulation, the state of the system remains constant between events, so one is not interested in taking into account this inactive time. As such, when an event has been "processed" (i.e., accounted for), the simulation time is incremented to the time of the next event and that event is then "processed."

There are two general approaches to the design of a discrete event simulator: (1) the event-based approach, and (2) the process-based approach.

The event-based approach focuses on events, which cause changes in the state of the system. Using such an approach – **which is what we will focus on for the purposes of this course** – events are the primary "objects" we set out to "code." Specifically, events throughout a modeled system are processed in time sequential order without consideration of boundaries of processes, etc. The process-based interaction approach focuses on how customers (e.g., processes, requests, packets, ... etc.) flow through the system. Using this approach, each entity in the system is represented by a separate process.

6.1.3 An Example Simulation

To illustrate discrete-event simulation using an event-based approach, let us take the very simple queuing system below, with just a single queue and a single server.

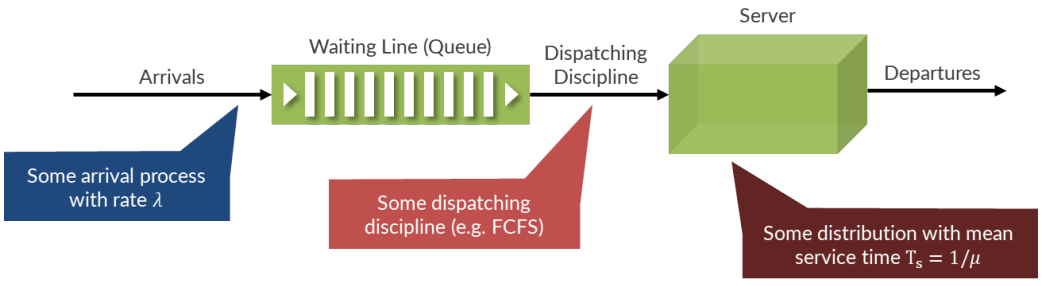

Figure 6.1: A simple queuing system with generic arrival, dispatching, and service laws.

Let's assume that requests to the above queue are best modeled as having interarrival times that are (say) uniformly distributed between 0 and 2 seconds (i.e., all arrival times between 0 and 2

seconds are equally likely). In addition, suppose that service times are best modeled by a uniformly distributed random variable with a range between 0.5 and 1.5 seconds (i.e., any service time between 0.5 and 1.5 seconds is equally likely). We will illustrate how this system can be evaluated using simulation. Conceptually we have two separate and independent random processes at work: the arrival process to the queue and the amount of time it will take each arrival to execute. Clearly, what will happen in the queue is the interaction between these two processes.

One way to "simulate" this interaction is to generate the arrival time and service time of each request and then to walk through these numbers to figure out how the queue in the above system will "evolve" over time. We do so next.

The first two columns in Table 6.1 show the Inter-Arrival Time (IAT) and the service time (T_s) that each customer in the system will experience. We generated these values to follow the uniform distributions for IAT and Ts as given above.

Suppose that now we consider our system at time zero ($t = 0$), with no customers in the system. Take the random values generated in the first two columns of Table 6.1 and let's ask ourselves the question: **What will happen next?**

The answer is that after **0.33 seconds** have elapsed the first customer will appear. The queue is empty and the server is idle so this customer can proceed directly to being served. **What will happen next?**

The answer is that after a further **0.56 seconds** have elapsed (i.e. at $t = 0.33 + 0.56 = 0.89$) that first customer will finish its execution and leave the system. **What will happen next?**

The answer is that at time $t = 1.5$ the second customer shows up. The queue is empty and the server is idle so this customer can proceed directly to being served. **What will happen next?**

The answer is that at time $t = 1.74$ a third customer will show up. At that time we have a customer in the queue, requiring the third customer to wait in the queue. **What will happen next?**

The answer is that at time $t = 2.07$, the second customer is finally done and service can start on the third customer. **What will happen next?**

The answer is that at time $t = 2.2$ a fourth customer will show up. At that time we have a customer in the queue, requiring the fourth customer to wait in the queue. **What will happen next?**

And so on —

You can hopefully see from the above how we are *artificially reproducing* (i.e., simulating) the operation of our queuing system by figuring out when various events occurred and how the state of the system changed as a result of each such event.

Notice that, as illustrated above, we are only looking at the state of the system at discrete points in time – namely when customers appear and when service ends. We only concerned ourselves with the discrete points $t = 0.33, 0.89, 1.50, 1.74, 2.07, 2.20, \ldots$ etc.

	IAT	T_s	Arrival	Service Start	Service End	T_q
			0.00			
	0.33	0.56	0.33	0.33	0.89	0.56
	1.17	0.57	1.50	1.50	2.07	0.57
	0.24	0.95	1.74	2.07	3.02	1.28
	0.45	0.83	2.20	3.02	3.85	1.66
	1.01	1.39	3.21	3.85	5.25	2.04
	1.02	0.51	4.22	5.25	5.75	1.53
	1.26	1.33	5.48	5.75	7.08	1.60
	0.73	1.45	6.21	7.08	8.53	2.32
	1.37	1.41	7.58	8.53	9.95	2.36
	1.86	0.52	9.44	9.95	10.46	1.02
	1.33	0.69	10.77	10.77	11.46	0.69
	0.72	1.40	11.49	11.49	12.88	1.40
	1.21	1.48	12.70	12.88	14.36	1.66
	1.81	0.62	14.51	14.51	15.13	0.62
	1.03	1.40	15.54	15.54	16.94	1.40
	1.51	1.45	17.05	17.05	18.50	1.45
	1.85	1.00	18.90	18.90	19.90	1.00
	0.54	1.24	19.43	19.90	21.13	1.70
	0.87	1.34	20.30	21.13	22.48	2.17
	0.35	0.64	20.65	22.48	23.11	2.46
	0.10	1.26	20.75	23.11	24.38	3.63
	1.26	0.62	22.01	24.38	25.00	2.99
	0.73	1.03	22.74	25.00	26.03	3.29
	1.73	1.29	24.47	26.03	27.32	2.85
Mean	**1.02**	**1.04**				**1.76**

Table 6.1: An example of simulation for a simple single-sever system.

The various columns in Table 6.1 show how the independently generated interarrival time (IAT) and service time (T_s) "interact" to result in various waiting times for customers and eventual response times T_q.

Once we have done a simulation such as shown above then we can easily calculate statistics about the system and/or answer questions about the system. For example, the average time a customer spends in the system T_q, is simply the difference between when service is done for a customer and when they arrived to the system.

We can also calculate statistics on the number of customers in the system. Here the system has 0 customers from $t = 0$ to $t = 0.33$, 1 customer from $t = 0.33$ to $t = 0.89$, 0 customers from $t = 0.89$ to $t = 1.5$, etc. Using this information the time-weighted average of the number of customers can be computed to be $q = [0 \cdot (0.33 - 0.00) + 1 \cdot (0.89 - 0.33) + 0 \cdot (1.5 - 0.89) + \ldots]/t_{max}$, where t_{max} represents the last observation time.

Box 6.1.1 On Steady State of Simulation Experiments

Note that the above calculations (both for average time in the system and average number of customer) took into account the system when we first started - when it was completely empty. This is probably biasing (rendering inaccurate) the statistics we are calculating and so it is common in simulation to allow some time to elapse (so the system "warms up") before starting to collect information for use in calculating summary statistics.

The question of how much "warm-up" time is enough is a difficult question to answer, but in general, one rule of thumb is that it should be at least as much as the desired duration of the experiment itself. More "scientifically" we may want to be confident that whatever statistics we have collected have stabilized. One way to do this is to compute confidence intervals (as will be discussed in following lectures) over different windows of time and stop when the confidence intervals seem to "stabilize".

6.2 Anatomy of a Simulator

In the above very simple example, we are able to simulate the behavior of the system by merely generating two sequences of random numbers and asking ourselves *what happens next?* along the way. This is doable for a world that consists of one state variable (the number of people in the queue) and two events (birth and death). Now consider a complicated system with hundreds of queues and possibly thousands of events. There could also be complicated processes for the selection of who to serve next from a queue[1] as well as complicated formulae for determining service times[2].

[1] In the above example, we assumed FCFS, but perhaps we may want to simulate a more sophisticated behavior for the scheduler!

[2] In the above example, we assumed that service time is the same for all requests – a uniformly generated random number. What if it is a different distribution for different "classes" of requests? What if the service time depends on the state of the system (e.g., who was served before, or how many requests are in the queue, etc.)

Clearly, asking *what happens next?* is not the way to go. What we need is a methodical way to build discrete event simulators. To do so, we look back at our simple example and try to figure out a generic "algorithm," restricting ourselves to the event-based model.

Model of the simulated system: The first question we must answer is what the "variables" are we wish to model in the underlying system. The answer to this depends on the level of details we would like our simulation to achieve. For instance, with reference to the above simple queue, if the purpose of the simulation is just to measure the average number of requests in the queue, then we can model that system using a single variable – namely q, the number of customers in the system. However, if the purpose of the simulation is to measure (say) the characteristics of the response time (e.g., mean and standard deviation) then modeling the system with a single variable q will not be enough. In particular, we would need to have individual records for all the requests currently in the queue (stored in some data structure, such as a linked list or array), with each record registering the arrival time of the request and when it started its service. This would allow us to deduce the response time for each request when service is done, which we then can log and eventually use to compute the characteristics of the response time through the above queue. For our illustrative example, let us assume that this is what we wish to retain in the model.

Events in the simulated system: The next step in constructing our simulator is the identification of the various possible events in our system. Typically, we would expect one event for every arrival process and one event for every server. Thus, in our illustrative example, we would worry about two events: a birth event (the arrival of a new customer to the queue) and the departure event (the completing of service for a customer).

Notice that in addition to events directly related to the simulated system in question, we may want to introduce events to help us "measure" that system. For instance, suppose that we are asked to plot the number of customers in the system (q) at some fixed time interval (say every 5 seconds). To do so, we could simply introduce an event whose only purpose is to take a snapshot of the system queue (e.g., counting the number of requests in the system), and logging the results (say in a file) for later processing/plotting/etc. Let's call such an event the "monitor" event.

Event specification: The last step in constructing our simulation model is the specification of how the events we identified in the previous step change the state of the simulated system. In the simple example at hand, we have to specify how the "birth," "death" and "logging" events affect the state of the simulated system.

- **Birth Event:** What happens upon a "birth"? What changes in the simulator's state would be triggered by a birth event? Clearly, we must add the newly born request to the data structure (say linked list) of requests in the system. In addition, if this request happens to be the only one in the system (i.e., the server was idle upon its birth) then we must "start service" for that request, which means that we can "predict" when service will be done (and hence schedule a death event), by generating a random "service time" according to the distribution of service times and adding that to the current time. The last thing that an arrival allows us to do is predict (and hence schedule) when the next birth will occur! Notice that we are given a distribution of inter-arrival times (IAT), which means that upon a birth we can predict when the next birth will occur by generating a random "IAT time" according to the specified distribution of IAT times, and adding that to the current time.

- **Death Event:** Now we turn our attention to what happens upon a death event. Clearly, we must remove the record of the request from the data structure (say linked list) of requests in the system. In doing so, we may also collect and compute some statistics (for example figure out the response time for the serviced request) and/or print some information, etc. What else? Upon the completion of service, the server should check if other requests are pending in the queue. If any are there, then the server must start service of the "next" request in the queue. This means that we can "predict" when service will be done, by generating a random "service time" according to the distribution of service times and adding that to the current time.
- **Monitor Event:** As we mentioned before, a monitor event allows us the opportunity to take a snapshot of the system at some regular intervals. As such, when such an event occurs, we could inspect the data structures describing the simulated system and log them, say by writing them to a file, etc. What else? Well, the occurrence of a monitor event should allow us to predict (and hence schedule) the next monitor event (just as a birth allowed us to predict when the next birth should happen).

The above three events can be "coded" into three functions that we need to execute every time the corresponding event occurs. But who is going to call these functions? What determines the order in which these events will be called? Clearly, the last thing we need is a "controller" of the simulation!

Simulation Control: The simulation controller is nothing more than an entity that (1) initializes the state of the simulated system, (2) keeps track of time, and (3) calls the above functions (which will change the state of the simulated system) at the right time – the "right time" being specified by a schedule dictated by when (and modified as a result of) various event occurrences in the simulated world. Let's target these three tasks, but in reverse order.

The simulation controller needs to keep a schedule (or calendar) of "future events." Such a schedule could be a simple linked list, with each entry representing a future event. The linked list would be kept sorted in time, so it would be always possible to ask "what is the next event" (a.k.a., what happens next) and also to schedule future events by inserting them into the linked list. The heap data structure (sorted binary tree) makes a good schedule, as it allows easy insertion and sorting of new events. Each entry in the schedule (linked list) would include the scheduled time for the event as well as an identification of the event so that we may be able to execute the function corresponding to that event at the scheduled time. In a language like `C`, the identification of the event could be a pointer to the "function" specifying the event.

The simulation controller needs to keep track of time. Notice, however, that the only times at which interesting things happen are the times of events – recall that in a discrete event simulation, nothing happens in the time between events! Thus, keeping track of time is nothing more than figuring out what the next event is, setting the current time to the time of that event, and executing that event since its time has come, and then repeating this for the next event, and the next event, etc. – a simple "while" loop until the desired simulation time expires!

Finally, initialization involves setting the current time (say to 0) and initializing the schedule. For our simple illustrative example (and often in more complex ones), this is nothing more than inserting a single birth into the schedule!

Notice that the simulation control we have just described is **generic** in the sense that it is **exactly the same for any simulated system!**

Indeed, the simulation controller code will look like the pseudo code given in Listing 6.1.

```
main() {
  state = initializeState();
  schedule = initializeSchedule();
  time = 0;
  while(time < maxTime) {
    event = getNextEvent(schedule);
    time = event.time;
    event.function(schedule, state, time);
  }
}
```

Listing 6.1: Structure of a generic simulation controller.

Box 6.2.1 Generating "random" numbers for simulation purposes

As I mentioned in class, any random number generated by a program is not "really" random by virtue of having a deterministic approach to generate it (i.e., the computer program/algorithm). For any deterministic program to generates a truly random number it must be infinite in size– hence the impossibility of generating such numbers from within a program. We call randomness that one gets by executing a program "pseudo randomness," and one can show that any program to generate pseudo random numbers must have a "cycle." Of course, the longer the cycle the better the generator.

Thus, programs (e.g., library functions such as `drand48()`) that generate random numbers can be thought of as walking through a cycle of (pseudo) random numbers. But, where in this cycle do we want to start? Well, to control that, functions used to generate pseudo random numbers allow a programmer to specify a "seed" for the generator, which essentially specifies where to start the cycle. One typically initializes the seed once as part of a program's initialization (e.g., at the beginning of a simulation). Seeding is an important functionality because it allows the programmer to use the same program (e.g., simulator) to get independent "runs" and hence be able to build confidence in the results he/she obtains from the simulator.

To understand this, we go through an example. Assume that the following are the values we obtain by successively calling a random number generator `rnd4()` with a cycle of $2^4 = 16$ using a given seed (typical generators have cycles that have at least that many zeros!).

$$5,7,1,9,2,1,6,3,2,8,0,1,5,3,8,4,5,7,1,9,2,1,6,3,2,8,0,1,5,3,8,4,\ldots$$

If we restart our program but use the exact same seed we used in the previous run, we would

get exactly the same sequence, i.e.,

$$5,7,1,9,2,1,6,3,2,8,0,1,5,3,8,4,5,7,1,9,2,1,6,3,2,8,0,1,5,3,8,4,\ldots$$

However, if we use a different seed, we may get a different sequence (with same period), for example:

$$6,3,2,8,0,1,5,3,8,4,5,7,1,9,2,1,6,3,2,8,0,1,5,3,8,4,5,7,1,9,2,1,\ldots$$

Thus, by using the same seed, a programmer could force a program to reproduce the same exact "random" outcomes (e.g., arrivals). This is very handy for debugging purposes, for example. More importantly, it gives a programmer the ability to reproduce their results without having to store all the random numbers generated in a run. A good practice, therefore, is to store (as part of the records of a simulation experiment) the seed used to obtain the results in that experiment.

Now, of course, to be able to run a simulation many times and obtain independent results, one has to use different seeds for every run. How do we automatically choose such seeds? Recall that we want all these seeds to be different (or else we will have some repeated experiments with identical results). One trick that programmers do is to use a function of the "Time of Day" or the least significant bits of some large counter (e.g., number of seconds elapsed since 1971!) The point is to use a seed that we are unlikely to use again in another one of the experiments.

Box 6.2.2 From Uniformly Distributed Random Numbers to General Distributions

The basic pseudo random number generator provided to a programmer produces a uniformly distributed random number – say a real number between 0 and 1. However, often times, we would want to generate random variables that follow different distributions.

Let's start with a simple case. Let's say that we want to generate a random variable x that has one of two values, $x = 0$ and $x = 1$, and that the probability of $x = 0$ is 0.70 and the probability of $x = 1$ is 0.30. Clearly, we can do this by generating a uniformly distributed random number Y. If the number Y is greater than or equal to 0.0 and less than 0.70, then we say $x = 0$, otherwise we say $x = 1$. Visually, we can present this with the diagram in Figure 6.2. Using a uniform (pseudo) random number generator, we get a number Y. If Y is between 0 and 0.70 we generate $x = 0$, and if Y is between 0.70 and 1.0, we generate $x = 1$.

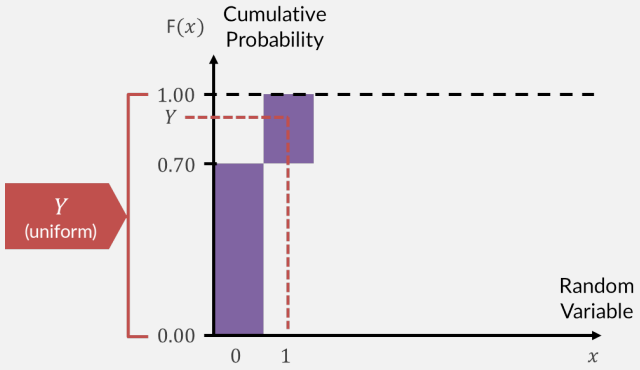

Figure 6.2: Translating an uniform sample Y into a sample distributed according to a custom distribution with two possible outcomes.

Clearly, we can generalize this for more than two outcomes. For example, the diagram in Figure 6.3 shows the same process for 5 different outcomes: 0, 1, 2, 3, and 4, with discrete probabilities: 0.3, 0.3, 0.2, 0.1, and 0.1, respectively. If Y is between 0 and 0.3, we say $x = 0$, if it is between 0.3 and 0.6, we say $x = 1$, if it is between 0.6, and 0.8, we say $x = 2$, if it is between 0.8 and 0.9, we say $x = 3$, and if it is between 0.9 and 1, we say $x = 4$. Notice that the values: 0.3, 0.6, 0.8, 0.9, and 1 are precisely the discrete values of the cumulative distribution function $F(x)$. Namely, $F(0) = 0.3$, $F(1) = 0.6$, $F(2) = 0.8$, $F(3) = 0.9$, and $F(4) = 1.0$.

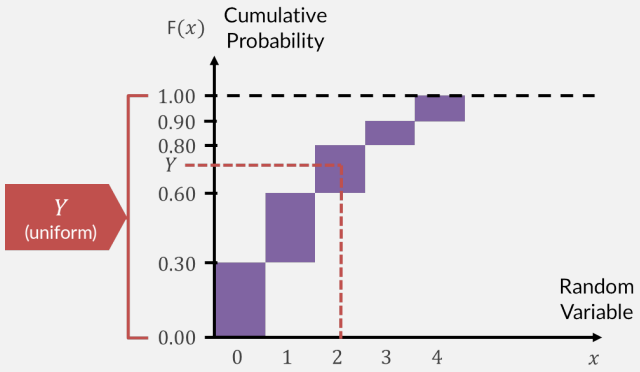

Figure 6.3: Translating an uniform sample Y into a sample distributed according to a custom distribution with 5 possible outcomes (0-4).

Thus, another way to describe the process is to say that we generate a uniformly random number Y which we compare to the cumulative probability distribution function $F(x)$ for the random variable we want to generate X. By solving the equation $F(x) \geq Y > F(x-1)$, we obtain the value of x that we should be generating. In the previous example, if $Y = 0.69$, then the random value of x that we would generate should be 2 because that is the only value of x that satisfies the equation $F(x) \geq 0.69 > F(x-1)$.

The above procedure works as well if the random variable we want to generate is continuous (as opposed to discrete). The only difference is that for continuous random variables we would

simply equate $F(x)$ to Y (rather than using inequalities).

Visually, we can present the process of using a uniformly generated pseudo random number to generate a continuous random variable with the diagram in Figure 6.4. Namely, we find the value of x that solves the equation $F(x) = Y$.

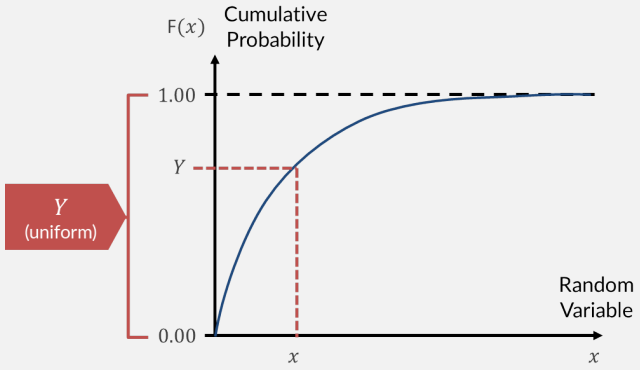

Figure 6.4: Translating an uniform sample Y into a sample distributed according to a continuous distribution (e.g. exponential distribution).

For example if we want to generate a random variable x that follows an exponential distribution, we would equate the cumulative distribution function for the exponential distribution, namely $F(x) = 1 - e^{-\lambda \cdot x}$, to Y and solving for x. In particular, we would get the following equation $Y = 1 - e^{-\lambda \cdot x}$, which when we solve for x we get: $x = -\ln(1-Y)/\lambda$

6.3 Simulation versus Analysis

The advantages of using simulation, as opposed to an analytic approach (e.g., using queuing analysis) are many, but mostly it is related to our ability to handle much more complex behaviors. Specifically, the mathematics of queuing analysis is hard and only valid for certain statistical distributions, whereas the mathematics of simulation is easy and can cope with any statistical distribution. Moreover, in simulations we can more easily deal with time-dependent (also known as non-stationary) behaviors. For example, assume that the service time for a given request depends (say) on the time of day, or on the state of some other queue. Analyzing such a behavior could be quite complicated and often not possible to carry on without significant approximations. Indeed, in some situations it is virtually impossible to build the equations that queuing analysis demands (e.g. for features like queue switching, queue dependent work rates, etc.).

One disadvantage of simulation is that it is difficult to find optimal solutions, unlike queuing analysis, for example, where we have equations involving parameters that we can simply optimize. The only way to attempt to optimize using simulation is to: (1) make a change, (2) run the simulation to see if an improvement has been achieved or not, and (3) repeat!