14 Event-Driven Simulation

Having covered how to generate random variables in the previous chapter, we are now in good shape to move on to the topic of creating an event-driven simulation. The goal of simulation is to predict the performance of a computer system under various workloads. A big part of simulation is modeling the computer system as a queueing network. Queueing networks will be revisited in much more detail in Chapter 27, where we *analytically* address questions of performance and stability (analysis is easier to do after covering Markov chains and hence is deferred until later).

For now, we only explain as much as we need to about queueing networks to enable simulation. We will start by discussing how to simulate a single queue.

14.1 Some Queueing Definitions

Figure 14.1 depicts a queue. The circle represents the **server** (you can think of this as a CPU). The red rectangles represent **jobs**. You can see that one of the jobs is currently being served (it is in the circle) and three other jobs are queueing, waiting to be served, while three more jobs have yet to arrive to the system. The red rectangles have different heights. The height of the rectangle is meant to represent the **size** of a job, where size indicates the job's service requirement (number of seconds needed to process the job). You can see that some jobs are large, while others are small. Once the job finishes **serving** (being processed) at the server, it leaves the system, and the next job starts serving. We assume that new jobs arrive over time. The time between arrivals is called the **interarrival time**. Unless otherwise stated, we assume that jobs are served in first-come-first-served (FCFS) order.

Question: If the arrival process to a queue is a Poisson process, what can we say about the interarrival times?

Answer: The interarrival times are independent and identically-distributed (i.i.d.) $\sim \text{Exp}(\lambda)$ where $\frac{1}{\lambda}$ represents the mean interarrival time and λ can be viewed as the **rate of arrivals** in jobs/s.

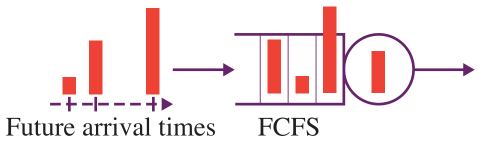

Figure 14.1 *Single queue with arrivals.*

We will generally assume a **stochastic setting** where all quantities are i.i.d. random variables. We will denote a job's size by the random variable (r.v.) S. For example, if $S \sim \text{Uniform}(0, 10)$, then jobs each require independent service times ranging between 0 and 10 seconds. The interval times between jobs is denoted by the r.v. I, where again we assume that these are independent. For example, if $I \sim \text{Exp}(\lambda)$, where $\lambda = 0.1$, then the average time between arrivals is 10 seconds. When running a simulation based on distributions for interarrival times and job sizes, we are assuming that these distributions are reasonable approximations of the **observed workloads** in the actual computer system being simulated.

However, it is also possible to assume that job sizes and interarrival times are taken from a **trace**. In that case, the simulation is often referred to as a **trace-driven simulation**. The trace typically includes information collected about the system over a long period of time, say a few months or a year.

Question: What are some advantages of using a trace to drive the simulation as opposed to generating inputs from distributions?

Answer: The trace captures correlations between successive interarrival times and/or successive job sizes. For example, it might be the case that a small job is more likely to be followed by another small job, or that arrivals tend to occur in bursts. This is harder to capture with independent random variables, although one can certainly try to create more complex probabilistic models of the workload [33].

We define the **response time of job**, typically denoted by r.v. T, to be the time from when the job first arrives until it completes service. We can also talk about the **waiting time** (a.k.a. **delay**) of a job, denoted by r.v. T_Q, which is the time from when the job first arrives until it first receives service. We define the **number of jobs in system**, denoted by r.v. N, to be the total number of jobs in the system. We define the **server utilization**, denoted by ρ, as the long-run fraction of time that the server is busy.

The goal of a simulation is typically to understand some aspect of the system performance. As an example, suppose that we are interested in the **mean response time**, $\mathbf{E}[T]$. We can think of this as follows. Let T_1 denote the response time of the first job, T_2 the response time of the second job, etc. Then,

$$\mathbf{E}[T] = \frac{1}{n}\sum_{i=1}^{n} T_i,$$

where it is assumed that n is sufficiently large that the mean response time is not changing very much. Thus, to get the mean response time, we can imagine having each of the first n jobs record its response time, where we then average over all of these.

14.2 How to Run a Simulation

Imagine that we want to simulate the queue shown in Figure 14.1, where the interarrival times are i.i.d. instances of r.v. I and the job sizes (service requirements) are i.i.d. instances of some r.v. S. Assume that we know how to generate instances of I and S using the techniques described in Chapter 13.

Question: Do we run this system in real time?

Answer: No, that would take forever.

The whole point is to be able to process millions of arrivals in just a few hours. To do this, we use an **event-driven simulation**. The idea is to maintain the **system state** at all times and also maintain a **global clock**. Then we ask,

"What is the next event that will cause a change in the system state?"

We then increase the time on the global clock by the time until this next event, and we update the system state to reflect the next event. We also update the times until the next events. We then repeat this process, stepping through events in near-zero time.

For example, let's consider an event-driven simulation of the queue in Figure 14.1.

Question: What is the system state?

Answer: The state is the current number of jobs in the system.

Question: What are events that change the state?

Hint: There are only two such events.

Answer: A new arrival or a job completion.

The interarrival times will need to be generated according to r.v. I. The job sizes (service requirements) will need to be generated according to r.v. S.

Question: Do we generate all the arrival times and all the job sizes for the whole simulation in advance and store these in a large array?

Answer: No, it's much simpler to generate these as we need them.

Let's run through how this works. We are going to maintain four variables:

1. Clock: represents the time;
2. State: represents the current number of jobs in the system;
3. Time-to-next-completion;
4. Time-to-next-arrival.

The simulation starts here: State is 0 jobs. Clock = 0. There's no job serving, so Time-to-next-completion = ∞. To determine the time to the next arrival, we generate an instance of I, let's say $I = 5.3$, and set Time-to-next-arrival = 5.3.

We ask which event will happen first. Since $\min(\infty, 5.3) = 5.3$, we know the next event is an arrival.

We now update everything as follows: State is 1 job. Note that this job starts serving immediately. Clock = 5.3. To determine the time to the next completion, we generate an instance of S representing the service time of the job in service, say $S = 10$, and set Time-to-next completion = 10. To determine the next arrival we generate an instance of I, say $I = 2$, and set Time-to-next-arrival = 2.

We again ask which event will happen first. Since $\min(10, 2) = 2$, we know the next event is an arrival.

We now update everything as follows: State is 2 jobs. Clock $= 5.3 + 2 = 7.3$. Time-to-next-completion $= 10 - 2 = 8$, because the job that was serving has completed 2 seconds out of its 10 second requirement. To determine the next arrival we generate an instance of I, say $I = 9.5$, and set Time-to-next-arrival = 9.5.

We again ask which event will happen first. Since $\min(8, 9.5) = 8$, we know the next event is a completion.

We now update everything as follows: State is 1 job. Clock $= 7.3 + 8 = 15.3$. To determine the time to the next completion, we generate an instance of S, say $S = 1$, and set Time-to-next-completion = 1. Time-to-next-arrival $= 9.5 - 8 = 1.5$ because 8 seconds have already passed since the last arrival, decreasing the previous time from 9.5 down to 1.5.

We continue in this manner, with updates to the state happening only at job arrival times or completions. Note that we only generate new instances of I or S as needed.

Question: When exactly do we generate a new instance of I?

Answer: There are two times: The main time we generate a new instance of I is immediately after a new job arrives. However, we also generate a new instance of I at the very start of the simulation when there are 0 jobs.

Question: When exactly do we generate a new instance of S?

Answer: The main time we generate a new instance of S is immediately after a job completes service. However, there is an exception to this rule, which occurs when the system moves from State 1 (one job) to State 0 (zero jobs). At that time, the Time-to-next-completion is set to ∞. Additionally, we generate a new instance of S at the time when the system moves from State 0 to State 1.

Question: What changes if a trace is used to provide interarrival times and/or job sizes – that is, we run a trace-driven simulation?

Answer: Nothing, really. The same approach is used, except that rather than generating a new instance of I or S when we need it, we just read the next value from the trace.

14.3 How to Get Performance Metrics from Your Simulation

So now you have your simulation running. How do you figure out the **mean response time**? We propose two methods, the first of which we already discussed briefly.

Method 1: Every job records the clock time when it arrives and then records the clock time when it completes. Taking the difference of these gives us the job's response time. We now just need to average the response time over all the jobs.

Question: Should we write each job's response time into a file and then take the average at the end of our simulation?

Answer: No, the writing wastes time in our simulation. You should be able to maintain a running average. Let $\overline{T}_n$ denote the average over the first n jobs:

$$\overline{T}_n = \frac{1}{n} \sum_{i=1}^{n} T_i .$$

Then $\overline{T}_{n+1}$ can easily be determined from $\overline{T}_n$ as follows:

$$\overline{T}_{n+1} = \frac{1}{n+1} \cdot \left(\overline{T}_n \cdot n + T_{n+1} \right).$$

Method 2: We perform several runs of the simulation. A single **run** involves running the simulation, without bothering to have jobs record their response time, until we get to the 10,000th job (we've picked this number arbitrarily). We then record the response time of that 10,000th job. We now start the simulation from scratch, repeating this process for, say, 1000 runs. Each run provides us with just a single number. We now take the average of all 1000 numbers to obtain the mean response time.

Question: What are some benefits to Method 1?

Answer: Method 1 is simpler because we don't have to keep restarting the simulation from scratch.

Question: What are some benefits to Method 2?

Answer: Method 2 provides independent measurements of response time. Notice that Method 1 does not provide independent measurements, because if a job has high response time then it is likely that the subsequent job also has high response time (the queue is currently long). Having independent measurements has the advantage that we can create a **confidence interval** around our measured mean response time. We defer discussion of how to obtain confidence intervals to Chapter 19.

If one runs a simulation for long enough, it really doesn't matter whether one uses Method 1 or Method 2, assuming that your system is well behaved.[1] This brings us to another question.

Question: How long is "long enough" to run a simulation?

Answer: We want to run the simulation until the metric of interest, in this case mean response time, appears to have stabilized (it's not going up or down substantially). There are many factors that increase the time it takes for a simulation to converge. These include load, number of servers, and any type of variability, either in the arrival process or the job service times. It is not uncommon to need to run a simulation with a billion arrivals before results stabilize.

Now suppose the goal is not the mean response time, but rather the **mean number** of jobs in the system, $\mathbf{E}[N]$. Specifically, we define the mean number as a time-

[1] Technically, by well behaved we mean that the system is "ergodic." It suffices that the system empties infinitely often. For a more detailed discussion of ergodicity, see Chapter 25 and Section 27.7.

average, as follows: Let $M(s)$ denote the number of jobs in the system at time s. Then,

$$\mathbf{E}[N] = \lim_{t\to\infty} \frac{\int_{s=0}^{s=t} M(s)ds}{t}. \tag{14.1}$$

Think of this as summing the number of jobs in the system over every moment of time s from $s = 0$ to $s = t$ and then dividing by t to create an average. Obviously we're not really going to take t to infinity in our simulation, but rather just some high enough number that the mean number of jobs stabilizes.

Question: But how do we get $\mathbf{E}[N]$ from our simulation? We're not going to look at the number at every single time s. Which times do we use? Can we simply measure the number of jobs in the system as seen by each arrival and average all of those?

Answer: This is an interesting question. It turns out that if the arrival process is a Poisson process, then we can simply record the number of jobs as seen by each arrival. This is due to a property called PASTA (Poisson arrivals see time averages), explained in [35, section 13.3]. Basically this works because of the memoryless property of a Poisson process, which says that the next arrival can come at any time, which can't in any way be predicted. Thus the arrival times of a Poisson process are good "random" points for sampling the current number of jobs.

Unfortunately, if the arrival process is not a Poisson process, then having each arrival track the number of jobs that it sees can lead to very wrong results.

Question: Can you provide an example for what goes wrong when we average over what arrivals see?

Answer: Suppose that $I \sim \text{Uniform}(1, 2)$. Suppose that $S = 1$. Then every arrival finds an empty system and thus we would conclude that the mean number of jobs is 0, when in reality the mean number of jobs is: $\frac{2}{3} \cdot 1 + \frac{1}{3} \cdot 0 = \frac{2}{3}$.

Question: So how do we measure the mean number of jobs in the system if the arrival process is *not* a Poisson process?

Answer: The easiest solution is to simulate a Poisson process (independent of the arrival process) and sample the number of jobs at the times of that simulated Poisson process. This adds more events since we now have arrivals, completions, and Poisson events.

14.4 More Complex Examples

We now turn to some more complex examples of queueing networks.

Example 14.1 (Router with finite buffer)

Figure 14.2 shows a router with finite (bounded) buffer space. There is room for $n = 6$ packets, one in service (being transmitted) and the others waiting to be transmitted. Note that all the packets are purposely depicted as having the same size, as is typical for packets. When a packet arrives and doesn't find space, it is dropped.

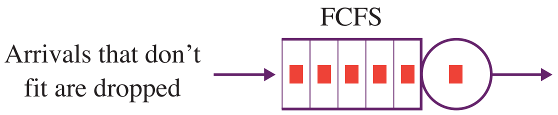

Figure 14.2 *Queue with finite buffer space.*

In terms of running the simulation, nothing changes. The system state is still the number of packets in the system. As before we generate packet sizes and interarrival times as needed. One of the common reasons to simulate a router with finite buffer space is to understand how the buffer space affects the fraction of packets that are dropped. We will investigate this in Exercise 14.4.

Question: Suppose we are trying to understand mean response time in the case of the router with finite buffer space. What do we do with the dropped packets?

Answer: Only the response times of packets that enter the system are counted.

Example 14.2 (Packet-routing network)

Figure 14.3 shows a network of three queues, where all queues are unbounded (infinite buffer space). A packet may enter either from queue 1 or from queue 2. If the packet enters at queue 2, it will serve at queue 2 and leave without joining any other queues. A packet entering at queue 1 will serve at queue 1 and then move to either queue 2 or queue 3, each with probability 0.5. We might be interested here in the response time of a packet entering at queue 1, where response time is the time from when the packet arrives at queue 1 until it leaves the network (either at server 2 or at server 3).

Question: What is the state space for Figure 14.3?

Answer: The system state is the number of packets at each of the three queues.

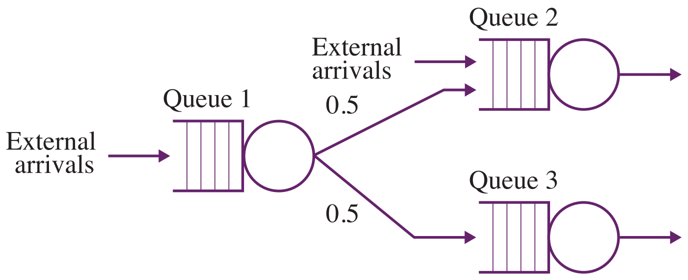

Figure 14.3 *Network of queues.*

Question: How many possible events do we need to watch for now?

Answer: We need to track five possible events. For queue 1, we need to track Time-to-next-arrival and Time-to-next-completion. For queue 3, we only need to track Time-to-next-completion. The arrival times at queue 3 are determined by flipping a fair coin after each completion at queue 1. Likewise, for queue 2, the internal arrival times at queue 2 are determined by flipping a fair coin after each completion at queue 1. However, queue 2 also has *external* arrivals. These external arrivals need to be tracked. Thus, for queue 2 we need to track the Time-to-next-external-arrival and Time-to-next-completion.

Example 14.3 (Call center)

Figure 14.4 shows an example of a call center, as might be operated by a company like Verizon. There is an arrival stream of incoming calls. There are k servers (operators) ready to accept calls. When a call comes in, it goes to any operator who is free (we imagine that all operators are homogeneous). If no operators are free, the call has to queue. Whenever an operator frees up, it takes the call at the head of the queue (i.e., calls are served in FCFS order). We assume that the service times of calls are i.i.d., represented by r.v. S. Here we might be interested in the average or variance of the queueing time experienced by calls.

Question: Do calls leave in the order that they arrived?

Answer: No. Calls enter service in the order that they arrived, but some calls might be shorter than others, and hence may leave sooner, even though they entered later.

Question: What is the state space for Figure 14.4?

Answer: The system state is the total number of jobs in the system (we do not need to differentiate between those in service and those queued), plus the remaining service time for each of the jobs in service.

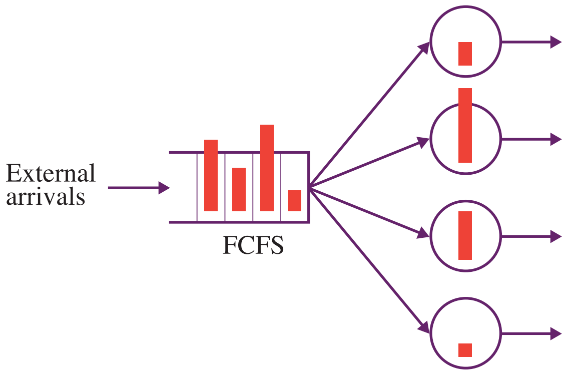

Figure 14.4 *Call center with $k = 4$ servers.*

Question: What are the events that we need to track?

Answer: We need to track $k + 1$ events. These are the Time-to-next-completion at each of the k servers and the Time-to-next-arrival for the system.

We will explore additional examples in the exercises.

14.5 Exercises

14.1 **Mean response time in an M/M/1 queue**
In this problem you will simulate a queue, as shown in Figure 14.1, and measure its mean job response time, $\mathbf{E}[T]$. Job sizes are i.i.d. instances of $S \sim \text{Exp}(\mu)$, where $\mu = 1$. The arrival process is a Poisson process with rate λ. The queue is called an M/M/1 to indicate that both interarrival times and job sizes are memoryless (M). For each value of $\lambda = 0.5, 0.6, 0.7, 0.8, 0.9$, record both $\mathbf{E}[T]$ and $\mathbf{E}[N]$. Draw curves showing what happens to $\mathbf{E}[T]$ and $\mathbf{E}[N]$ as you increase λ. To check that your simulation is correct, it helps to verify that Little's Law ($\mathbf{E}[N] = \lambda \cdot \mathbf{E}[T]$) holds. Little's Law will be covered in Chapter 27.

14.2 **Server utilization of an M/M/1 queue**
Repeat Exercise 14.1, but this time measure the server utilization, ρ, which is the long-run fraction of time that the server is busy. To get ρ, you will sample the server at the times of job arrivals to determine the average fraction of arrivals that see a busy server.

14.3 **Doubling the arrival rate and the service rate**
Repeat Exercises 14.1 and 14.2, but this time double each of the original

arrival rates and simultaneously double the service rate. Specifically, in these “new” runs, our arrival rates will be: $\lambda = 1.0, 1.2, 1.4, 1.6, 1.8$, and our job sizes will be i.i.d. instances of $S \sim \text{Exp}(\mu)$, where $\mu = 2$.
(a) How does ρ_{new} compare with ρ_{orig}?
(b) How does $\mathbf{E}[N_{\text{new}}]$ compare with $\mathbf{E}\left[N_{\text{orig}}\right]$?
(c) How does $\mathbf{E}[T_{\text{new}}]$ compare with $\mathbf{E}\left[T_{\text{orig}}\right]$?
Try to provide intuition for your findings. [Hint: Think about how doubling the arrival rate and service rate affects time scales.]

14.4 **Effect on loss probability of various improvements**
As in Exercise 14.1, we have a queue whose arrivals are a Poisson process with rate λ, and whose job sizes are i.i.d. instances of r.v. $S \sim \text{Exp}(\mu)$. Let $\mu = 1$ and $\lambda = 0.9$. Now suppose that the queue is bounded so that at most $n = 5$ jobs can be in the system (one serving and the other four queueing).
(a) Simulate the system to determine the loss probability, namely the fraction of arriving jobs that are dropped because they don’t fit.
(b) That loss probability is deemed too high, and you are told that you must lower it. You are considering two possible improvements:
  (i) Double the capacity of your system by setting $n = 10$.
  (ii) Double the speed of your server (double μ) to $\mu = 2$.
  Which is more effective at reducing loss probability? Simulate and find out.
(c) Conjecture on why you got the answer that you got for part (b). Do you think that your answer to part (b) is always true? If you can’t decide, run some more simulations with different values of λ.

14.5 **Effect of variability of job size on response time**
In this problem, we will study the effect of variability of job sizes on response time by using a DegenerateHyperexponential(μ, p) distribution, which will allow us to increase the variability in job size, S, by playing with μ and p parameters. The **Degenerate Hyperexponential** with parameters μ and p is defined as follows:

$$S \sim \begin{cases} \text{Exp}(p\mu) & \text{w/prob } p \\ 0 & \text{w/prob } 1-p \end{cases}.$$

(a) What is $\mathbf{E}[S]$? Is this affected by p?
(b) What is the squared coefficient of variation of S, namely C_S^2?
(c) What is the range of possible values for C_S^2, over $0 < p < 1$?
(d) Create a simulation to determine mean queueing time, $\mathbf{E}\left[T_Q\right]$, in a single queue. The arrival process to the queue is a Poisson process with rate $\lambda = 0.8$. The job sizes are denoted by $S \sim$ DegenerateHyperexponential$(\mu = 1, p)$. You will run multiple simulations, each with the appropriate value of p to create the cases of $C_S^2 = 1, 3, 5, 7, 9$. Draw a graph with $\mathbf{E}\left[T_Q\right]$ on the y-axis and C_S^2 on

the x-axis. Note that a job of size 0 may still experience a queueing time, even though its service time is 0.

(e) What happens when C_S^2 increases? Why do you think this is? Think about it from the perspective of the time that the average job waits.

14.6 **Favoring short jobs over long ones**

Consider a queue with a Poisson arrival process and mean interarrival time of 110. Job sizes are drawn i.i.d. from

$$S \sim \begin{cases} 1 & \text{w/prob } 0.5 \\ 200 & \text{w/prob } 0.5 \end{cases}.$$

(a) Simulate the queue where the jobs are served in FCFS order. What is the mean response time, $\mathbf{E}[T]$?

(b) Now consider a different scheduling policy, called Non-Preemptive Shortest Job First (NP-SJF). Like FCFS, NP-SJF is *non-preemptive*, meaning that once we start running a job, we always finish it. However, in NP-SJF the jobs of size 1 always have priority over the jobs of size 200. Specifically, NP-SJF maintains two FCFS queues, one with jobs of size 1 and the other with jobs of size 200, where, whenever the server is free, it picks to run the job at the head of the queue of jobs of size 1. Only if there is no job of size 1 does the server run a job of size 200. Note that among jobs of a given size, the service order is still FCFS. Simulate NP-SJF and report $\mathbf{E}[T]$. Try to use as little state as you can get away with.

(c) You should find that $\mathbf{E}[T]$ is much lower under NP-SJF scheduling than under FCFS scheduling. Why do you think this is?

14.7 **SRPT queue versus FCFS queue**

The Shortest Remaining Processing Time (SRPT) scheduling policy minimizes mean response time, $\mathbf{E}[T]$ [66, 67]. Under SRPT, at all times the server is working on that job with the shortest *remaining* processing time. The SRPT policy is *preemptive*, meaning that jobs can be stopped and restarted with no overhead. Under SRPT, a new arrival will preempt the current job serving if and only if the new arrival has size which is smaller than the remaining time on the job in service.

(a) Suppose we have an SRPT queue and job j is currently running. Can job j be preempted by any of the other jobs currently in the queue?

(b) In an SRPT queue, can there be multiple jobs which have each received partial service so far?

(c) Simulate an SRPT queue, with Poisson arrival process with rate $\lambda = 0.45$. Assume that the job sizes are $S \sim \text{BoundedPareto}(k = 0.004, p = 1000, \alpha = 0.5)$ (see Definition 10.5). What is $\mathbf{E}[T]^{\text{SRPT}}$?

(d) Perform the same simulation but for a FCFS queue. What is $\mathbf{E}[T]^{\text{FCFS}}$?

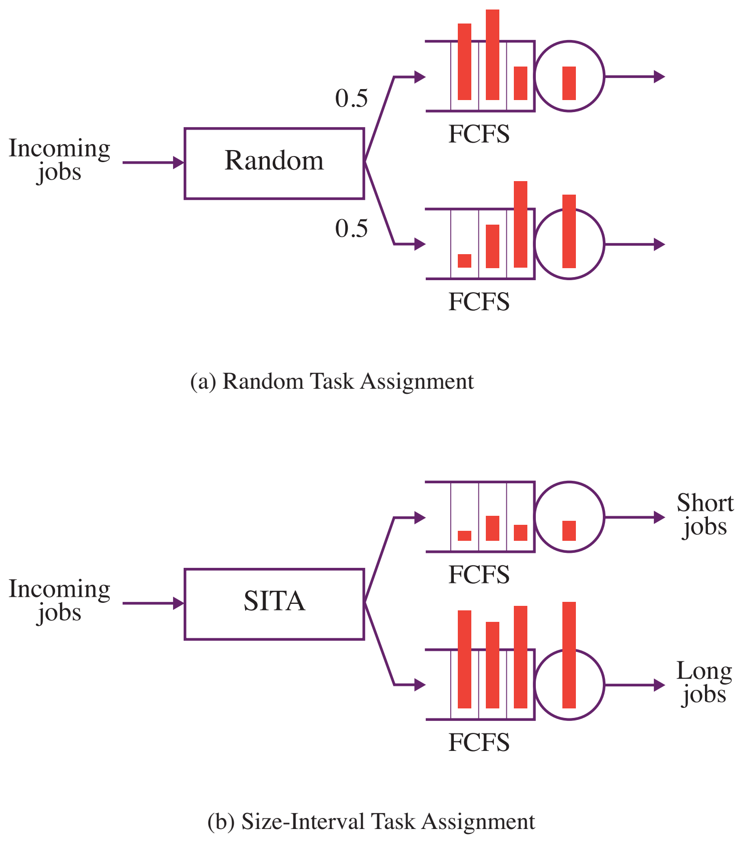

Figure 14.5 *Random dispatching versus SITA dispatching.*

14.8 **Size-Interval Task Assignment versus Random**
Figure 14.5 illustrates a server farm with two identical FCFS queues. In Figure 14.5(a), every incoming arrival is dispatched (assigned) with probability 0.5 to the first queue and probability 0.5 to the second queue. This is called Random task assignment (Random). In Figure 14.5(b), if the incoming arrival is "small" then it is dispatched to the top queue, and if it is "large" it is dispatched to the bottom queue. This is called Size-Interval Task Assignment (SITA) [36, 34]. Suppose that arrivals occur according to a Poisson process with rate $\lambda = 0.5$. Assume that job sizes are i.i.d. and follow a BoundedPareto($k = 0.004, p = 1000, \alpha = 0.5$) distribution (see Definition 10.5) with mean $\mathbf{E}[S] = 2$.
(a) Simulate Random assignment and report the mean queueing time $\mathbf{E}\left[T_Q\right]$.
(b) Simulate SITA. Use a size cutoff of 58.3, where jobs smaller than this cutoff are deemed "small." Report $\mathbf{E}\left[T_Q\right]$.
(c) Which was better? Why do you think this is?