

In this section we will present a general *next-event* approach to building discrete-event simulation models. From this chapter on, this next-event approach will be the basis for all the discrete-event simulation models developed in this book.

The motivation for considering the next-event approach to discrete-event simulation is provided by considering the relative complexity of the effort required to extend the discrete-event simulation models in Section 3.1 to accommodate the slightly more sophisticated corresponding models in Section 3.3. That is, at the computational level compare the simplicity of program `ssq2` in Section 3.1 with the increased complexity of the extension to `ssq2` that would be required to reproduce the results in Example 3.3.2. Yet the only increase in the complexity of the associated single-server service node model is the addition of immediate feedback. Similarly, compare the simplicity of program `sis2` in Section 3.1 with the increased complexity of the extension to `sis2` that would be required to reproduce the results in Example 3.3.4. Yet in this case the only increase in the complexity of the associated simple inventory system model is the addition of a delivery lag.

5.1.1 DEFINITIONS AND TERMINOLOGY

While programs `ssq2` and `sis2` and their corresponding extensions in Section 3.3 are valid and meaningful (albeit simple) discrete-event simulation programs, they do not adapt easily to increased model complexity and they do not generalize well to other systems. Based on these observations we see the need for a more general approach to discrete-event simulation that applies to queuing systems, inventory systems and a variety of other systems as well. This more general approach — next-event simulation — is based on some important definitions and terminology: (1) system state, (2) events, (3) simulation clock, (4) event scheduling, and (5) event list (calendar).

System State

Definition 5.1.1 The *state* of a system is a complete characterization of the system at an instance in time — a comprehensive “snapshot” in time. To the extent that the state of a system can be characterized by assigning values to variables, then *state variables* are what is used for this purpose.

To build a discrete-event simulation model using the next-event approach, the focus is on refining a description of the state of the system and its evolution in time. At the *conceptual* model level the state of a system exists only in the abstract as a collection of possible answers to the following questions: what are the state variables, how are they interrelated, and how do they evolve in time? At the *specification* level the state of the system exists as a collection of mathematical variables (the state variables) together with equations and logic describing how the state variables are interrelated and an algorithm for computing their interaction and evolution in time. At the *computational* level the state of the system exists as a collection of program variables that collectively characterize the system and are systematically updated as (simulated) time evolves.

Example 5.1.1 A natural way to describe the state of a single-server service node is to use the number of jobs in the service node as a state variable. As demonstrated later in this section, by refining this system state description we can construct a next-event simulation model for a single-server service node with or without immediate feedback.

Example 5.1.2 Similarly, a natural way to describe the state of a simple inventory system is to use the current inventory level and the amount of inventory on order (if any) as state variables. As demonstrated in the next section, by refining this system state description we can construct a next-event simulation model of a simple inventory system with or without delivery lag.

Events

Definition 5.1.2 An *event* is an occurrence that may change the state of the system. By definition, the state of the system can only change at an event time. Each event has an associated *event type*.

Example 5.1.3 For a single-server service node model with or without immediate feedback, there are two types of events: the *arrival* of a job and the *completion of service* for a job. These two types of occurrences are events because they have the potential to change the state of the system. An arrival will always increase the number in the service node by one; if there is no feedback, a completion of service will always decrease the number in the service node by one. When there is feedback, a completion *may* decrease the number in the service node by one. In this case there are two event types because the “arrival” event type and the “completion of service” event type are not the same.

Example 5.1.4 For a simple inventory system with delivery lag there are three event types: the occurrence of a *demand* instance, an *inventory review*, and the *arrival* of an inventory replenishment order. These are events because they have the potential to change the state of the system: a demand will decrease the inventory level by one, an inventory review may increase the amount of inventory on order, and the arrival of an inventory replenishment order will increase the inventory level and decrease the amount of inventory on order.

The *may* in Definition 5.1.2 is important; it is not necessary for an event to cause a change in the state of the system, as illustrated in the following four examples: (1) events can be scheduled that statistically *sample*, but do not change, the state of a system, (2) for a single-server service node with immediate feedback, a job’s completion of service will only change the state of the system if the job is not fed back, (3) for a single-server service node, an event may be scheduled at a prescribed time (e.g., 5 PM) to cut off the stream of arriving jobs to the node, which will not change the state of the system, and (4) for a simple inventory system with delivery lag, an inventory review will only change the state of the system if an order is placed.

Simulation Clock

Because a discrete-event simulation model is dynamic, as the simulated system evolves it is necessary to keep track of the current value of simulated time. In the implementation phase of a next-event simulation, the natural way keep track of simulated time is with a floating point variable, which is typically named `t`, `time`, `tnow`, or `clock` in discrete-event simulation packages. The two examples that follow the definition of the simulation clock highlight the inability of the discrete-event simulation approach to easily generalize or embellish models. The next-event framework overcomes this limitation.

Definition 5.1.3 The variable that represents the current value of simulated time in a next-event simulation model is called the *simulation clock*.

Example 5.1.5 The discrete-event simulation model that program `ssq2` represents is heavily dependent on the job processing order imposed by the FIFO queue discipline. Therefore, it is difficult to extend the model to account for immediate feedback, or a finite service node capacity, or a priority queue discipline. In part, the reason for this difficulty is that there are effectively *two* simulation clocks with one coupled to the arrival events and the other coupled to the completion of service events. These two clocks are not synchronized and so it is difficult to reason about the temporal order of events if arrivals and completions of service are merged by feedback.

Example 5.1.6 The discrete-event simulation model that program `sis2` represents has only one type of event, inventory review, and events of this type occur deterministically at the beginning of each time interval. There is a simulation clock, but it is *integer-valued* and so is primitive at best. Because the simulation clock is integer-valued we are essentially forced to ignore the individual demand instances that occur within each time interval. Instead, all the demands per time interval are aggregated into one random variable. This aggregation makes for a computationally efficient discrete-event simulation program, but forces us in return to do some calculus to derive equations for the time-averaged holding and shortage levels. As outlined in Section 3.3, when there is a delivery lag the derivation of those equations is a significant task.

Event Scheduling

In a discrete-event simulation model it is necessary to use a *time-advance mechanism* to guarantee that events occur in the correct order — that is, to guarantee that the simulation clock never runs backward. The primary time-advance mechanism used in discrete-event simulation is known as *next-event* time advance; this mechanism is typically used in conjunction with *event scheduling*.

Definition 5.1.4 If event scheduling is used with a next-event time-advance mechanism as the basis for developing a discrete-event simulation model, the result is called a *next-event* simulation model.

To construct a next-event simulation model, three things must be done:

- construct a set of state variables that together provide a complete system description;
- identify the system event types;
- construct a collection of algorithms that define the state changes that will take place when each type of event occurs.

The model is constructed so as to cause the (simulated) system to evolve in (simulated) time by executing the events in increasing order of their scheduled time of occurrence. Time does not flow continuously; instead, the simulation clock is advanced discontinuously from event time to event time. At the computational level, the simulation clock is frozen during the execution of each state-change algorithm so that each change of state, no matter how computationally complex, occurs instantaneously relative to the simulation clock.

Event List

Definition 5.1.5 The data structure that represents the scheduled time of occurrence for the next possible event of each type is called the *event list* or *calendar*.

The event list is often, but not necessarily, represented as a priority queue sorted by the next scheduled time of occurrence for each event type.

5.1.2 NEXT-EVENT SIMULATION

Algorithm 5.1.1 A next-event simulation model consists of the following four steps:

- **Initialize.** The simulation clock is initialized (usually to zero) and, by looking ahead, the first time of occurrence of each possible event type is determined and scheduled, thereby initializing the event list.
- **Process current event.** The event list is scanned to determine the *most imminent* possible event, the simulation clock is then advanced to this event's scheduled time of occurrence, and the state of the system is updated to account for the occurrence of this event. This event is known as the "current" event.
- **Schedule new events.** New events (if any) that may be spawned by the current event are placed on the event list (typically in chronological order).
- **Terminate.** The process of advancing the simulation clock from one event time to the next continues until some terminal condition is satisfied. This terminal condition may be specified as a *pseudo*-event that only occurs once, at the end of the simulation, with the specification based on processing a fixed number of events, exceeding a fixed simulation clock time, or estimating an output measure to a prescribed precision.

The next-event simulation model initializes once at the beginning of a simulation replication, then alternates between the second step (processing the current event) and third step (scheduling subsequent events) until some terminate criteria is encountered.

Because event times are typically random, the simulation clock runs *asynchronously*. Moreover, because state changes *only* occur at event times, periods of system inactivity are ignored by advancing the simulation clock from event time to event time. Compared to the alternate, which is a *fixed-increment* time-advance mechanism, there is a clear computational efficiency advantage to this type of asynchronous next-event processing.*

In the remainder of this section, next-event simulation will be illustrated by constructing a next-event model of a single-server service node. Additional illustrations are provided in the next section by constructing next-event simulation models of a simple inventory system with delivery lag and a multi-server service node.

5.1.3 SINGLE-SERVER SERVICE NODE

The state variable $l(t)$ provides a *complete* characterization of the state of a single-server service node in the sense that

$$\begin{aligned} l(t) = 0 &\iff q(t) = 0 && \text{and} && x(t) = 0 \\ l(t) > 0 &\iff q(t) = l(t) - 1 && \text{and} && x(t) = 1 \end{aligned}$$

where $l(t)$, $q(t)$, and $x(t)$ represent the number in the node, in the queue, and in service respectively at time $t > 0$. In words, if the number in the service node is known, then the number in the queue and the status (idle or busy) of the server is also known. Given that the state of the system is characterized by $l(t)$, we then ask what events can cause $l(t)$ to change? The answer is that there are two such events: (1) an *arrival* in which case $l(t)$ is increased by 1; and (2) a *completion of service* in which case $l(t)$ is decreased by 1. Therefore, our conceptual model of a single-server service node consists of the state variable $l(t)$ and two associated event types, arrival and completion of service.

To turn this next-event conceptual model into a specification model, three additional assumptions must be made.

- The *initial state* $l(0)$ can have any non-negative integer value. It is common, however, to assume that $l(0) = 0$, often referred to as “empty and idle” in reference to the initial queue condition and server status, respectively. Therefore, the first event must be an arrival.

* Note that asynchronous next-event processing cannot be used if there is a need at the computational level for the simulation program to interact synchronously with some other process. For example, because of the need to interact with a person, so called “real time” or “person-in-the-loop” simulation programs must use a fixed-increment time-advance mechanism. In this case the underlying system model is usually based on a system of ordinary differential equations. In any case, fixed-increment time-advance simulation models are outside the scope of this book, but are included in some of the languages surveyed in Appendix A.

- Although the *terminal state* can also have any non-negative integer value, it is common to assume, as we will do, that the terminal state is also idle. Rather than specifying the number of jobs processed, our stopping criteria will be specified in terms of a time τ beyond which no new jobs can arrive. This assumption effectively “closes the door” at time τ but allows the system to continue operation until all jobs have been completely served. This would be the case, for instance, at an ice cream shop that closes at a particular hour, but allows remaining customers to be served. Therefore, the last event must be a completion of service.*
- Some mechanism must be used to denote an event as *impossible*. One way to do this is to structure the event list so that it contains *possible* events only. This is particularly desirable if the number of event types is large. As an alternate, if the number of event types is not large then the event list can be structured so that it contains both possible and impossible events — but with a numeric constant “ ∞ ” used for an event time to denote the impossibility of an event. For simplicity, this alternate event list structure is used in Algorithm 5.1.2.

To complete the development of a specification model, the following notation is used. The next-event specification model is then sufficiently simple that we can write Algorithm 5.1.2 directly.

- The simulation clock (current time) is t .
- The terminal (“close the door”) time is τ .
- The next scheduled arrival time is t_a .
- The next scheduled service completion time is t_c .
- The number in the node (state variable) is l .

The genius and allure of both discrete-event and next-event simulation is apparent, for example, in the generation of arrival times in Algorithm 5.1.2. The naive approach of generating and storing all arrivals prior to the execution of the simulation is not necessary. Even if this naive approach were taken, the modeler would be beset by the dual problems of memory consumption and not knowing how many arrivals to schedule. Next-event simulation simply primes the pump by scheduling the first arrival in the initialization phase, then schedules each subsequent arrival while processing the current arrival. Meanwhile, service completions weave their way into the event list at the appropriate moments in order to provide the appropriate sequencing of arrivals and service completions.

At the end of this section we will discuss how to extend Algorithm 5.1.2 to account for several model extensions: immediate feedback, alternative queue disciplines, finite capacity, and random sampling.

* The simulation will terminate at $t = \tau$ only if $l(\tau) = 0$. If instead $l(\tau) > 0$ then the simulation will terminate at $t > \tau$ because additional time will be required to complete service on the jobs in the service node.

Algorithm 5.1.2 This algorithm is a next-event simulation of a FIFO single-server service node with infinite capacity. The service node begins and ends in an empty and idle state. The algorithm presumes the existence of two functions `GetArrival` and `GetService` that return a random value of arrival time and service time respectively.

```

l = 0;                               /* initialize the system state */
t = 0.0;                              /* initialize the system clock */
ta = GetArrival();                 /* initialize the event list */
tc = ∞;                             /* initialize the event list */
while ((ta < τ) or (l > 0)) {        /* check for terminal condition */
    t = min(ta, tc);                /* scan the event list */
    if (t == ta) {                    /* process an arrival */
        l++;
        ta = GetArrival();
        if (ta > τ)
            ta = ∞;
        if (l == 1)
            tc = t + GetService();
    }
    else {                               /* process a completion of service */
        l--;
        if (l > 0)
            tc = t + GetService();
        else
            tc = ∞;
    }
}

```

If the service node is to be an M/M/1 queue (exponential interarrival and service times with a single server) with arrival rate 1.0 and service rate 1.25, for example, the two $t_a = \text{GetArrival}()$ statements can be replaced with $t_a = t + \text{Exponential}(1.0)$ and the two $t_c = \text{GetService}()$ statements can be replaced with $t_c = t + \text{Exponential}(0.8)$. The `GetArrival` and `GetService` functions can draw their values from a file (a “trace-driven” approach) or generate variates to model these stochastic elements of the service node.

Because there are just two event types, arrival and completion of service, the event list in Algorithm 5.1.2 contains at most two elements, t_a and t_c . Given that the event list is small and its size is bounded (by 2), there is no need for any special data structure to represent it. If the event list were larger, an array or structure would be a natural choice. The only drawback to storing the event list as t_a and t_c is the need to specify the arbitrary numeric constant “∞” to denote the impossibility of an event. In practice, ∞ can be any number that is *much* larger than the terminal time τ (100τ is used in program `ssq3`).

If the event list is large and its size is dynamic then a dynamic data structure is required with careful attention paid to its organization. This is necessary because the event list is scanned and updated each time an event occurs. Efficient algorithms for the insertion and deletion of events on the event list can impact the computational time required to execute the next-event simulation model. Henriksen (1983) indicates that for telecommunications system models, the choice between an efficient and inefficient event list processing algorithm can produce a five-fold difference in total processing time. Further discussion of data structures and algorithms associated with event lists is postponed to Section 5.3.

Program `ssq3`

Program `ssq3` is based on Algorithm 5.1.2. Note, in particular, the state variable `number` which represents $l(t)$, the number in the service node at time t , and the important time management structure `t` that contains:

- the event list `t.arrival` and `t.completion` (t_a and t_c from Algorithm 5.1.2);
- the simulation clock `t.current` (t from Algorithm 5.1.2);
- the next event time `t.next` ($\min(t_a, t_c)$ from Algorithm 5.1.2);
- the last arrival time `t.last`.

Event list management is trivial. The event type (arrival or a completion of service) of the next event is determined by the statement `t.next = Min(t.arrival, t.completion)`.

Note also that a statistics gathering structure `area` is used to calculate the time-averaged number in the node, queue, and service. These statistics are calculated exactly by accumulating time integrals via summation, which is valid because $l(\cdot)$, $q(\cdot)$, and $x(\cdot)$ are piecewise constant functions and only change value at an event time (see Section 4.1). The structure `area` contains:

- $\int_0^t l(s) ds$ evaluated as `area.node`;
- $\int_0^t q(s) ds$ evaluated as `area.queue`;
- $\int_0^t x(s) ds$ evaluated as `area.service`.

Program `ssq3` does not accumulate job-averaged statistics. Instead, the job-averaged statistics \bar{w} , \bar{d} , and \bar{s} are computed from the time-averaged statistics \bar{l} , \bar{q} , and \bar{x} by using the equations in Theorem 1.2.1. The average interarrival time \bar{r} is computed from the equation in Definition 1.2.4 by using the variable `t.last`. If it were not for the use of the assignment `t.arrival = INFINITY` to “close the door”, \bar{r} could be computed from the terminal value of `t.arrival`, thereby eliminating the need for `t.last`.

World Views and Synchronization

Programs `ssq2` and `ssq3` simulate exactly the same system. The programs work in different ways, however, with one clear consequence being that `ssq2` naturally produces *job*-averaged statistics and `ssq3` naturally produces *time*-averaged statistics. In the jargon of discrete-event simulation the two programs are said to be based upon different *world views*.^{*} In particular, program `ssq2` is based upon a *process-interaction* world view and program `ssq3` is based upon an *event-scheduling* world view. Although other world views are sometimes advocated, process interaction and event-scheduling are the two most common. Of these two, event-scheduling is the discrete-event simulation world view of choice in this and all the remaining chapters.

Because programs `ssq2` and `ssq3` simulate exactly the same system, these programs should be able to produce exactly the same output statistics. Getting them to do so, however, requires that both programs process exactly the same stochastic source of arriving jobs and associated service requirements. Because the arrival times a_i and service times s_i are ultimately produced by calls to `Random`, some thought is required to provide this synchronization. That is, the random variates in program `ssq2` are always generated in the alternating order $a_1, s_1, a_2, s_2, \dots$ while the order in which these random variates are generated in `ssq3` cannot be known a priori. The best way to produce this synchronization is to use the library `rngs`, as is done in program `ssq3`. In Exercise 5.1.3, you are asked to modify program `ssq2` to use the library `rngs` and, in that way, verify that the two programs can produce exactly the same output.

5.1.4 MODEL EXTENSIONS

We close this section by discussing how to modify program `ssq3` to accommodate several important model extensions. For each of the four extensions you are encouraged to consider what would be required to extend program `ssq2` correspondingly.

Immediate Feedback

Given the function `GetFeedback` from Section 3.3, we can modify program `ssq3` to account for immediate feedback by just adding an `if` statement so that `index` and `number` are not changed if a feedback occurs following a completion of service, as illustrated.

```

else {
    if (GetFeedback() == 0) {
        index++;
        number--;
    }
    /* process a completion of service */
    /* this statement is new */

```

^{*} A world view is the collection of concepts and views that guide the development of a simulation model. World views are also known as *conceptual frameworks*, *simulation strategies*, and *formalisms*.

For a job that is not fed back, the counter for the number of departed jobs (`index`) is incremented and the counter for the current number of jobs in the service node (`number`) is decremented. The simplicity of the immediate feedback modification is a compelling example of how well next-event simulation models accommodate model extensions.

Alternate Queue Disciplines

Program `ssq3` can be modified to simulate *any* queue discipline. To do so, it is necessary to add a dynamic queue data structure such as, for example a singly-linked list where each list node contains the arrival time and service time for a job in the queue, as illustrated.*

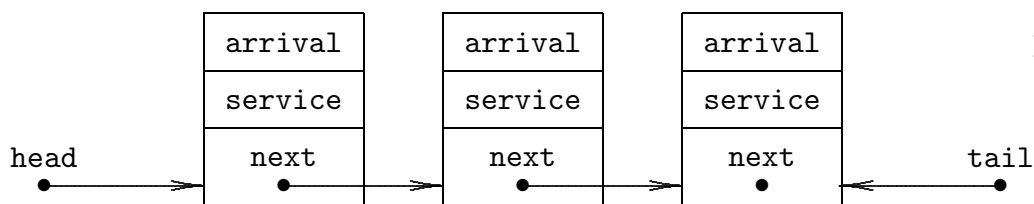


Figure 5.1.1.
Queue
data
structure.

Two supporting queue functions, `Enqueue` and `Dequeue`, are also needed to insert and delete jobs from the queue, respectively. Then `ssq3` can be modified as follows:

- use `Enqueue` each time an arrival event occurs and the server is busy;
- use `Dequeue` each time a completion of service event occurs and the queue is not empty.

The details of this important modification are left as an exercise. This modification will result in a program that can be tested for correctness by using a FIFO queue discipline and reproducing results from program `ssq3`.

Note in particular that this modification can be combined with the immediate feedback modification illustrated previously. In this case, the `arrival` field in the linked list would hold the time of feedback for those jobs that are fed back. The resulting program would allow a priority queue discipline to be used for fed back jobs if (as is common) a priority assumption is appropriate.

Finite Service Node Capacity

Program `ssq3` can be modified to account for a finite capacity by defining a constant `CAPACITY` which represents the service node capacity (one more than the queue capacity) and declaring an integer variable `reject` which counts rejected jobs. Then all that is required is a modification to the “process an arrival” portion of the program, as illustrated.

* If this queue data structure is used, then service times are computed and stored at the time of arrival. In this way, each job’s delay in the queue and wait in the service node can be computed at the time of entry into service, thereby eliminating the need to compute job-averaged statistics from time-averaged statistics.

```

if (t.current == t.arrival) {                               /* process an arrival */
    if (number < CAPACITY) {
        number++;
        if (number == 1)
            t.completion = t.current + GetService();
    }
    else
        reject++;
    t.arrival = GetArrival();
    if (t.arrival > STOP) {
        t.last = t.current;
        t.arrival = INFINITY;
    }
}
}

```

This code replaces the code in program `ssq3` for processing an arrival. As with the immediate feedback modification, again we see the simplicity of this modification is a compelling example of how well next-event simulation models accommodate model extensions.

Random Sampling

An important feature of program `ssq3` is that its structure facilitates direct *sampling* of the current number in the service node or queue. This is easily accomplished by adding a sampling time element, say `t.sample`, to the event list and constructing an associated algorithm to process the samples as they are acquired. Sampling times can then be scheduled *deterministically*, every δ time units, or *at random* by generating sampling times with an *Exponential*(δ) random variate inter-sample time. In either case, the details of this modification are left as an exercise.

5.1.5 EXERCISES

Exercise 5.1.1 Consider a next-event simulation model of a three-server service node with a single queue and three servers. (a) What variable(s) are appropriate to describe the system state? (b) Define appropriate events for the simulation. (c) What is the maximum length of the event list for the simulation? (Answer with and without considering the pseudo-event for termination of the replication.)

Exercise 5.1.2 Consider a next-event simulation model of three single-server service nodes in series. (a) What variable(s) are appropriate to describe the system state? (b) Define appropriate events for the simulation. (c) What is the maximum length of the event list for the simulation? (Answer with and without considering the pseudo-event for termination of the replication.)

Exercise 5.1.3 (a) Use the library `rngs` to verify that programs `ssq2` and `ssq3` can produce *exactly* the same results. (b) Comment on the value of this as a consistency check for both programs.

Exercise 5.1.4 Add a sampling capability to program `ssq3`. (a) With deterministic inter-sample time $\delta = 1.0$, sample the number in the service node and compare the average of these samples with the value of \bar{l} computed by the program. (b) With average inter-sample time $\delta = 1.0$, sample *at random* the number in the service node and compare the average of these samples with the value of \bar{l} computed by the program. (c) Comment.

Exercise 5.1.5 Modify program `ssq3` by adding a FIFO queue data structure. Verify that this modified program and `ssq3` produce *exactly* the same results.

Exercise 5.1.6^a As a continuation of Exercise 5.1.5, simulate a single-server service node for which the server uses a shortest-job-first priority queue discipline based upon a knowledge of the service time for each job in the queue. (a) Generate a histogram of the wait in the node for the first 10000 jobs if interarrival times are *Exponential*(1.0) and service times are *Exponential*(0.8). (b) How does this histogram compare with the corresponding histogram generated when the queue discipline is FIFO? (c) Comment.

Exercise 5.1.7 Modify program `ssq3` to reproduce the feedback results obtained in Section 3.3.

Exercise 5.1.8 Modify program `ssq3` to account for a finite service node capacity. (a) Determine the proportion of rejected jobs for capacities of 1, 2, 3, 4, 5, and 6. (b) Repeat this experiment if the service time distribution is *Uniform*(1.0, 3.0). (c) Comment. (Use a large value of STOP.)

Exercise 5.1.9 (a) Construct a next-event simulation model of a single-server machine shop. (b) Compare your program with the program `ssms` and verify that, with a proper use of the library `rngs`, the two programs can produce *exactly* the same output. (c) Comment on the value of this as a consistency check for both programs.

Exercise 5.1.10^a An M/M/1 queue can be characterized by the following system state change mechanism, where the system state is $l(t)$, the number of jobs in the node:

- The transition from state j to state $j + 1$ is exponential with rate $\lambda_1 > 0$ (the arrival rate) for $j = 0, 1, \dots$
- The transition from state j to state $j - 1$ is exponential with rate $\lambda_2 > 0$ (the service rate) for $j = 1, 2, \dots$

If the current state of the system is some positive integer j , what is the probability that the next transition will be to state $j + 1$?