Next-Event Simulation

Lawrence M. Leemis and Stephen K. Park, Discrete-Event Simulation - A First Course, Prentice Hall, 2006

Hui Chen

Computer Science
Virginia State University
Petersburg, Virginia

March 16, 2016

Table of Contents

Motivation

- Making small modifications to our simple discrete-event simulations is non-trivial
  - Add feedback to *ssq2*
  - Add delivery lag to *sis2*
- Next-event simulation is a *more general approach* to discrete-event simulation, based on,
  - System state
  - Events
  - Simulation clocks
  - Event scheduling
  - Event list

System State

- The state of a system is a complete characterization of the system at an *instance* in time
  - Conceptual model: abstract collection of variables and how they evolve over time
  - Specification model: collection of mathematical variables together with logic and equations
  - Computational model: collection of program variables systematically updated
- Example 5.1.1: state of *ssq* is the *number of jobs in the node*
- Example 5.1.2: state of *sis* is current *inventory level* and the *amount of inventory on order* (if any)

Events

- An event is an occurrence that may change the state of the system
- Example 5.1.3: For *ssq*, events are *arrivals* or *completion* of a jobs
  - An *arrival* will *always* increase the *number of jobs in the node* by 1
  - If there is *no feedback*, a *completion* of a job will *always* decrease the number of jobs in the node by 1
  - With feedback, a *completion* of a job *may* decrease the *number of jobs in the node* by 1
- Example 5.1.4: For *sis* with delivery lag, events are *demand* instances, *inventory reviews*, and *arrival of inventory replenishment orders*
  - A demand will decrease the *inventory level* by 1
  - An inventory review might lead to an increase in the *amount of inventory on order*
  - The arrival of an order will increase the *inventory level* and decrease the *amount of inventory on order*.
- We can also define *artificial events*, e.g.,
  - Statistically sample the state of the system
  - Schedule an event at a prescribed time

Simulation Clock

- The *simulation clock* represents the current value of simulated time
- Previously introduced discrete-event simulation models lack definitive simulated time
  - As a result, it is difficult to generalize or embellish models
  - Example 5.1.5: It is hard to reason about *ssq2* because there are effectively two simulation clocks
    - *Arrival times* and *completion times* are not synchronized
    - It is difficult to reason about the temporal order of events if arrivals are merged by feedback with completion of service.
  - Example 5.1.6: In *sis2*, the only event is inventory review
    - The simulation clock is integer-valued and we have to aggregate all demand and to do some calculus to derive equations for the time-averaged holding and shortage levels
    - When there is a delivery lag that happens in a non-integer-valued time, the derivation of those equations is a significant task. (see Example 3.3.3 or Exercise 7-2)

Event Scheduling

- It is necessary to use a *time-advance* mechanism to guarantee that events occur in the correct order
- *Next-event* time advance is typically used in discrete-event simulation
- To build a next-event simulation model:
  - construct a set of state variables
  - identify the event types
  - construct a set of algorithms that define state changes for each event type
- The simulated system evolves in simulated time by executing the events in increasing order of their scheduled time of occurrence.
  - Simulation clock is advanced discontinuously from event time to event time

Event List

- The *event list* (or *calendar* is the data structure containing the time of next occurrence for each event type
- The event list is often, but not necessarily, represented as a priority queue sorted by the next scheduled time of occurrence of each event type
- More detailed discussion in the examples and a later discussion on event-list management.

Next-Event Simulation

Algorithm 5.1.1

1. Initialize.
   1.1 set simulation clock (usually to zero)
   1.2 set first time of occurrence for each event type
2. Process current event.
   2.1 scan event list to determine most imminent event
   2.2 advance simulation clock
   2.3 update state
3. Schedule new events
   3.1 The current event may spawn new events. The new events, if any, are placed in the event list
   3.2 The algorithm returns to step 2 if not terminated as in step 4
4. Terminate
   - Continue advancing the clock and handling events until termination condition is satisfied

Next-Event Simulation

- The simulation clock runs asynchronously; inactive periods are ignored
  - Simulation clock is advanced discontinuously from event time to event time
- Clearly, a computational advantage over fixed-increment time-advance mechanism

Next-Event Simualtion by Examples

- Single-Server Service Node
  - Model extension: immediate feedback
  - Model extension: alternative queue disciplines
  - Model extension: finite service node capacity
  - Model extension: random sampling
- Simple inventory system with delivery lag and random demand
- Multi-server service node

Single-Server Service Node: Concept Model

- The state variable $l(t)$ (the number of jobs at time t at the node) provides a complete characterization of the state of a *ssq*

$$l(t) = 0 \iff q(t) = 0 \quad \textit{and} \quad x(t) = 0$$

$$l(t) > 0 \iff q(t) = l(t) - 1 \quad \textit{and} \quad x(t) = 1$$

- Two events cause this variable to change
  1. An arrival causes $l(t)$ to increase by 1
  2. A completion of service causes $l(t)$ to decrease by 1

Single-Server Service Node: Specification Model

- The initial state $l(0)$ can have any non-negative value, typically 0
- The terminal state can be any non-negative value
  - Assume at time τ arrival process stopped. Remaining jobs processed before termination
- Some mechanism must be used to denote an event impossible
  - Only store possible events in event list
  - Denote impossible events with event time of ∞

Single-Server Service Node: Specification Model

- 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

Note the following,

- It is not necessary to generate and store all arrivals prior to the execution of the simulation.
- It only nees to schedule the 1st arrival in the intialization phase and then to schedule each subsequent arrival while processing the current arrival
- It inserts completion event in the event list.

Single-Server Service Node: Computational Model

Algorithm 5.1.2

```
/* 1. Initialize */
l = 0;
t = 0.0;
t_a = GetArrival(); /* initialize the event list */
t_c = ∞;

while ((t_a < τ) or (l > 0)) { /* 4. Terminate or not */
  /* process current event */
  /* 2.1 - 2.2 scan the event list and advance simulation time */
  t = min(t_a, t_c);
  if (t == t_a) { /* process an arrival */
    l ++;
    /* 3. schedule new arrival event */
    t_a = GetArrival();
    if (t_a > τ)
      t_a = ∞;
    /* 3. schedule new service event */
    if (l == 1)
      t_c = t + GetService();
  } else { /* process a completion */
    l - -;
    /* 3. schedule new service event */
    if (l > 0)
      t_c = t + GetService();
    else
      t_c = ∞;
  }
}
```

Program *ssq3*

- In *ssq3*, *long* variable *number* represents $l(t)$ and structure variable t represents time
  - the event list *t.arrival* and *t.completion*, i.e., t_a and t_c from Algorithm 5.1.2;
  - the simulation clock *t.current*, i.e., t from Algorithm 5.1.2;
  - the next event time *t.next*, i.e., $min(t_a, t_c)$ from Algorithm 5.1.2
  - the last arrival time *t.last*
- Time-averaged statistics are gathered with the structure variable *area*
  - $\int_0^t l(s)ds$ evaluated as *area.node*
  - $\int_0^t q(s)ds$ evaluated as *area.queue*
  - $\int_0^t x(s)ds$ evaluated as *area.service*

World Views and Synchronization

- Programs *ssq2* and *ssq3* simulate exactly the same system
- The two have different world views
  - *ssq2*: *process-interaction* world view, naturally produces job-averaged statistics
  - *ssq3*: *event-scheduling* world view, naturally produces time-averaged statistics

  The *event-scheduling* world view is the discrete-event simulation world view of cohice for the rest of the discussion.
- The programs should produce exactly the same statistics
  - Both requires requires *rngs*

Model Extension: Immediate Feedback

It is simple to accomodate *immediate feedback* in the next-event simulation model.

Immediate Feedback

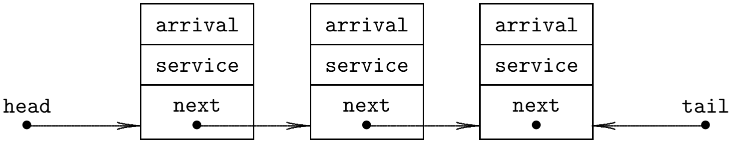
```
else { /* process a completion of service */
    if (GetFeedback() == 0) { /* this statement is new */
        index ++;
        number --;
    }
}
```

Exericse L10-1

You are to extend the *ssq3* program with *immediate feedback*. You may find *ssq3* in Blackboard. The instructor also provides a solution to Exercise L7-2 (Examples 3.3.2 only) and the program is *ssq2v2*. Complete the following.

1. Extend *ssq3* to accommodate *immediate feedback* as outlined in slide 18.
2. Adjust the parameters in the revised *ssq3* to match those in *ssq2v2*. Compare the output from the revised *ssq3* program to that of *ssq2v2* by (a) graphing the result in a single figure and (b) computing and graphing the relative difference of the two outputs in a second figure.
3. Observe both *ssq2v2* and the revised *ssq3*. Explain briefly the difference between the revision on *ssq2* (to obtain *ssq2v2*) and that on *ssq3* (to obtain the revised *ssq3*)
4. Submit the revised *ssq3*, the graphs, and your answer to bullet 3 above to Blackboard.

Model Extension: Alternate Queue Discipline



Program *ssq3* can be modified to simulate *any* queue discipline.

- Need to add a dynamic-queue data structure.
- Example: singly linked list.
  - Each list node contains the arrival time and service time for a job in the queue
  - Use *Enqueue* each time an arrival event occurs and the server is busy.
  - *Dequeue* each time a completion-of-service event occurs and the queue is not empty.
- Can be combined with with the immediate-feedback modification.
  - The *arrival* field in the linked list would hold the time of feeback for those fedback jobs.

Model Extension: Finite Service Node Capacity

Previously, assume the queue has *infinite* capacity. Program *ssq3* can be modified to account for a finite capacity.

Finite Service Node Capacity

```
if (t.current == t.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;
    }
}
```

Exericse L10-2

You are to extend the *ssq3* program with *finite capacity*. You may find *ssq3* in Blackboard. Complete the following.

1. Extend *ssq3* to accommodate *finite capacity* as outlined in slide 21.
2. What consistency check have you performed? What are the results?
3. Graph utilization versus capacity
4. What is the maximum queue capacity needed if we do not want to reject any jobs?
5. Submit the revised *ssq3*, the graphs, and your answers to bullet 2 and 4 above to Blackboard.

Model Extension: Random Sampling

- The structure of *ssq3* facilitates adding sampling
- Add a sampling event to the event list
  - Sample deterministically, every δ time units
  - Sample Randomly, every $\textit{Exponential}(\delta)$ time units

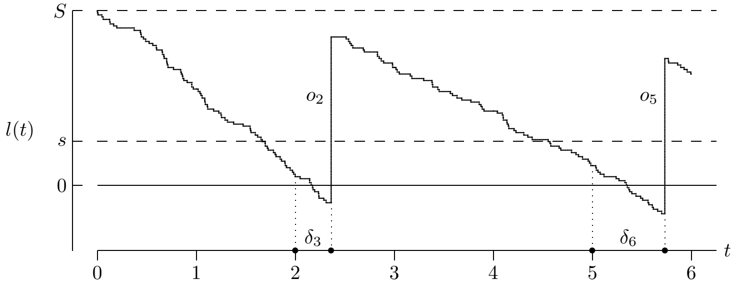
Simple Inventory System with Delivery Lag and Random Demand

Two changes relative to *sis2*

- *Uniform*$(0, 1)$ lag between inventory review and order delivery
- More realistic demand model
  - Demand instances for a single item occur at random
  - Average rate is λ demand instances per time interval
  - Time between demand instances is *Exponential*$(1/\lambda)$

Demand Models: A Comparison

- *sis2* used an aggregate demand for each time interval, generated as an *Equilikely*$(10, 50)$ random variate
  - Aggregate demand per time interval is random
  - Within an interval, time between demand instances is constant
  - Example: if aggregate demand is 25, inter-demand time is 0.04
- Now using *Exponential*$(1/\lambda)$ inter-demand times
  - Demand is modeled as an arrival process
  - Average demand per time interval is λ



Specification Model: States and Notation

- The simulation clock is t (real-valued)
- The terminal time is τ (integer-valued)
- Current inventory level is $l(t)$ (integer-valued)
- Amount of inventory on order, if any, is $o(t)$ (integer-valued). Necessary due to delivery lag
- $l(t)$ and $o(t)$ provide complete state description
- Initial state is assumed to be $l(0) = S$ and $o(0) = 0$
- Terminal state is assumed to be $l(\tau) = S$ and $o(\tau) = 0$
- Cost to bring $l(t)$ to S at simulation end (with no lag) must be included in accumulated statistics

Specification Model: Events

Three types of events can change the system state

- A demand for an item at time t. $l(t)$ decreases by 1.
- An inventory review at integer-valued time t
  - If $l(t) \geq s$, then $o(t) = 0$
  - If $l(t) < s$, then $o(t) = S l(t)$
- An arrival of an inventory replenishment order at time t
  - $l(t)$ increases by $o(t)$
  - $o(t)$ becomes 0

Algorithm 5.2.1: Initialization

- Time variables used for event list:
  - t_d: next scheduled inventory demand
  - t_r: next scheduled inventory review
  - t_a: next scheduled inventory arrival
- *infty* denotes impossible events

Initialization Step of Algorithm 5.2.1

I = S; /* initialize inventory level */
o = 0; /* initialize amount on order */
t = 0.0; /* initialize simulation clock */
t_d = GetDemand(); /* initialize event list */
t_r = t + 1.0; /* initialize event list */
$t_a = \infty$; /* initialize event list */

Algorithm 5.2.1: Main Loop

Main Loop of Algorithm 5.2.1

```
while (t < τ) {
  t = min(t_d, t_r, t_a); /* scan the event list */
  if (t == t_d) { /* process an inventory demand */
    l - -;
    t_d = GetDemand();
  } else if (t == t_r ) { /* process an inventory review */
    if (l < s) {
      o = S - l;
      end = GetLag();
      t_a = t + end;
    }
    t_r += 1.0;
  } else { /* process an inventory arrival */
    l += o;
    o = 0;
    t_a = 1;
  }
}
```

Program *sis3*

Implements Algorithm 5.2.1

- t.demand, t.review and t.arrive correspond to t_d , t_r, t_a
- State variables inventory and order correspond to $l(t)$ and $o(t)$
- sum.hold and sum.short accumulate the time-integrated holding and shortage integrals

A Multi-Server Service Node

- The single-server service node is extended to support multiple servers
- It is a natural generalization
  - Multi-server service nodes have both practical and theoretical importance
  - The event list size depends on the number of servers
    - For large numbers of servers, the event list data structure becomes important
  - Extensions of the multi-server node (immediate feedback, finite capacity, non-FIFO) are left as exercises

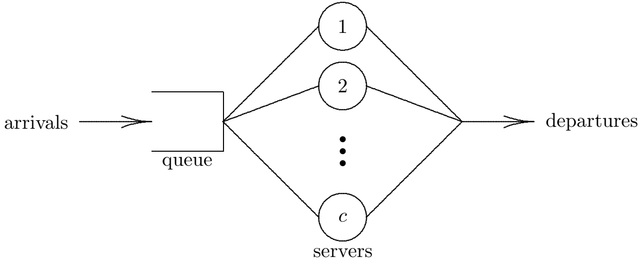
Conceptual Model

Definition 5.2.1: A multi-server service node consists of

- A single queue (if any)
- Two or more servers operating in parallel

At any instant in time,

- Each server is either busy or idle
- The queue is either empty or not empty
- If one or more servers is idle, the queue must be empty
- If the queue is not empty, all servers must be busy



Conceptual Model

- When a job arrives
  - If all servers are busy, the job enters the queue
  - Else an idle server is selected and the job enters service
- When a job departs a server
  - If the queue is empty, the server becomes idle
  - Else a job is removed from the queue, served by server Servers process jobs independently

Server Selection

- Definition 5.2.2: The algorithm used to select an idle server is called the *server selection rule*
- Common selection rules
  - Random selection: at random from the idle servers
  - Selection in order: lowest-numbered idle server
  - Cyclic selection: first available, starting after last selected (circular search may be required)
  - Equity selection: use longest-idle or lowest-utilized
  - Priority selection: choose the best idle server (modeler specifies how to dermine best)
- Random, cyclic, equity: designed to achieve equal utilizations
- If servers are statistically identical and independent, the selection rule has no effect on average performance of the service node
- The *statistically identical assumption* is useful for mathematicians; unnecessary for discrete-event simulation

Specification Model: States and Notation

- Servers in a multi-server service node are called service channels
  - c is the number of servers (channels)
  - The *server index* is $s = 1, 2, \dots, c$
- $l(t)$ denotes the number of jobs in the service node at time t
  - $l(t) \geq c$, all servers are busy and $q(t) = l(t)c$
  - If $l(t) < c$, some servers are idle
  - If servers are distinct, need to know which servers are idle
- For $s = 1, 2, \dots, c$, define $x_s(t)$: the number of jobs in service (0 or 1) at server s at time t
- The complete state description is $l(t), x_1(t), x_2(t), \dots, x_c(t)$

$$q(t) = l(t) - \sum_{s=1}^{c} x_s(t)$$

Specification Model: Events

What types of events can change state variables $l(t), x1(t), x2(t), \ldots, x_c(t)$?

- An arrival at time t
  - $l(t)$ increases by 1
  - If $l(t) \leq c$, an idle server s is selected, and $x_s(t)$ becomes 1
  - Else all servers are busy
- A completion of service by server s at time t
  - $l(t)$ decreases by 1
  - If $l(t) \geq c$, a job is selected from the queue to enter service
  - Else $x_s(t)$ becomes 0

There are $c + 1$ event types

Specification Model: Additional Assumption

- The initial state is an empty node
  - $l(0) = 0$
  - $x_1(0) = x_2(0) = \ldots = x_c(0) = 0$
  - The first event must be an arrival
- The arrival process is turned off at time τ
  - The node continues operation after time τ until empty
  - The terminal state is an empty node
  - The last event is a completion of service
- For simplicity, all servers are independent and statistically identical
- Equity selection is the server selection rule

All of these assumptions can be relaxed

Specification Model: Event List

0	t	x	arrival
1	t	x	completion of service by server 1
2	t	x	completion of service by server 2
3	t	x	completion of service by server 3
4	t	x	completion of service by server 4

- Can be organized as an array of $c + 1$ event types
- Field t: scheduled time of next occurrence for the event
- Field x: current activity status of the event
  - Superior alternative to using 1 to denote impossible events
  - For 0th event type, x denotes if arrival process is on or off
  - For other event types, x denotes if server is busy or idle
- For large c, consider alternate event-list structures (see later discussion)

Program *msq*

Implements this next-event multi-server service node simulation model

- State variable $l(t)$ is number
- State variables $x_1(t), x_2(t), \ldots, x_c(t)$ are part of the event list
- Time-integrated statistic $\int_0^t l(\theta)d\theta$
- Array sum records for each server
  - the sum of service times
  - the number served
- Function NextEvent searches the event list to find the next event
- Function FindOne searches the event list to find the longest-idle server (because equity selection is used)

Event-List Management

Some next-event simulations can have a great number of events on their event list simultaneously.

- An event list is the data structure that contains a list of events scheduled to occur in the future.
- The list is not necessarily sorted by the scheduled time of occurrence.
- Also called calendar, future events chain, sequencing set, future event set, ...
- Event lists are also called future events, event notices, transactions, records, ...

Event-list management is important

- Many next-event simulation models spend more CPU time on managing the event list than on any other aspects of the simulations.

Event-List Management: 4 Categories

2 boolean classifications

- fixed maximum or variable maximum number of events on the event list
- devised for one specific model or for a general-purpose simulation language

Based on the 2 boolean classifications, there are 4 categories of event-list management

No.	Number of Events	Model
1	fixed maximum	specific model
2	fixed maximum	general-purpose
3	variable maximum	specific model
4	variable maximum	general-purpose

Event-List Management: Operations

2 critical operations

- Insertion (*enqueue* or scheduling)
- Deletion (*dequeue*)

Additional operation

- Change Operation: change of an existing event
- Examine Operation: searches for an existing event
- Count: determine the number of events on the list

Event-List Management Criteria

3 criteria are used to assess the effectiveness of the data structure and algorithms for an event-management scheme

- Speed
- Robustness
- Adaptability

Event-List Management by Example

Consider the timesharing computer system model (Henriksen 1983) to discuss event-list management schemes.

- Simulation models for the timesharing computer system
  - Concept, specification, and computational models
- Event-list management schemes
  - Using *array*
  - Using a single *linked list*
  - Using multiple *linked lists*
  - Using *binary trees*
  - Using *heaps*
  - Hybrid schemes

Summary

- A generic approach to discrete-event simulation: next-event simulation
- Examples of next-event simulations
- Event-list management by examples