# CX 6010 Assignment 3: Discrete Event Simulation

**Due Dates:**

- Due: 11:00 AM, Friday, October 13, 2017
- Revision (optional): 11:55 PM, Monday October 16, 2017

You will work with one other student to develop a discrete event simulation program and use it to analyze a system. Discrete event simulations are widely used in science and engineering applications in areas such as manufacturing, supply chains, transportation, telecommunications, particle physics, business operations, etc. Rather than using a time-stepped approach, the simulation advances from one *event* to the next, where each event represents something "interesting" occurring in the actual system, i.e., the *physical system*. For example, typical events might be the arrival of a new customer in a simulation of a department store or a vehicle arriving at an intersection in a traffic simulation. Each event contains a timestamp, a simulation time value indicating when that event occurs in the physical system. The timestamp is analogous to the time step number in a time-stepped simulation, however, events occur at irregular points in time, not at regular, periodic time steps. As such, time is usually represented by a floating point number in a discrete event simulation, and simulation time advances at irregular intervals as the computation proceeds from one event to the next.

The simulation includes a number of state variables that represent the current stat of the system, e.g., the number of customers waiting in line or the state of a traffic signal. The computation consists of a sequence of event computations. Each event computation can (1) modify one or more state variables, and/or (2) schedule one or more new events into the simulated future. For example, when modeling air traffic at an airport, the computation for an event denoting that an aircraft has just touched down on the runway might schedule a new event five minutes into the simulated future to represent the aircraft arriving at the arrival gate and beginning to unload passengers.

The simulation program contains two main part:

- *Simulation engine*. This part of the program is independent of the simulation application, i.e., the same simulation engine software could be used to simulate a transportation system or a manufacturing application. It includes a data structure called the *future event list* (FEL) that is a priority queue that contains the set of events that have been scheduled, but have not yet been processed. The simulation engine holds the main *event processing loop* which repeatedly (1) removes the smallest timestamped event from the FEL, and (2) calls a function (defined in the simulation application, discussed next) to simulate that event. The loop continues processing events until some termination condition is met, e.g., simulation time reaches a certain value. The simulation maintains a *clock* variable indicating how far the simulation has advanced in simulation time. It also includes a function called by the simulation application to schedule a new event, i.e., to allocate memory for the event, fill in various parameters, and insert the new event into the FEL. The main event processing loop updates the clock variable in each iteration of the loop to the timestamp of the event it just removed from the FEL.
- *Simulation application*. This part of the program contains code to model the physical system. It includes a set of state variables that represent the current state of the system being modeled. It also includes one or more functions or *event handler* procedures, one for each type of event modeled by the simulation. The event handler procedures collectively model the operation of the system being simulated. To develop the simulation application,

one must define the state variables and the different types of events that are modeled, and implement a function for each different event type. The simulation application also includes a number of variables used to collect statistics concerning the simulation, e.g., the average time an aircraft must wait to land at an airport.

You will work with your partner to jointly develop the simulation program. One individual is responsible for the simulation application, and the other for the simulation engine. The interface (application program interface, or API) between the two must be clearly defined and documented in a header (`.h`) file. The simulation engine and application must reside in different files.

A sample discrete event simulation program is provided to help you develop your code. You may reuse any part of this example program to complete this assignment.

The following describes the simulation application and engine in more detail. A variety of parameters and specific values are defined below. Although your program will only use the values specified below, your code should be written so any of these can be easily modified.

**The Simulation Application**

The simulation application models the operation of a call center that handles both incoming and outgoing telephone calls. Incoming calls are typically questions about a product while outgoing calls are sales calls attempting to sell more product. The call center includes two types of employees (agents), termed "service" and "sales" agents. Service agents are *only* able to handle incoming calls. Sales agents are able to handle both incoming and outgoing calls. Assume your call center has 700 service agents and 300 sales agents. Your simulation should be written to model several hours of operation in the call center.

Incoming calls are assumed to follow a Poisson distribution with average arrival rate of `A` calls per second. This means the time between successive calls is exponentially distributed with a mean value of `1.0/A` seconds. Each incoming call is immediately routed to an agent. The system first tries to route the call to a service agent. If none are available, it tries to route the call to a sales agent. If there are no agents of any type available, the call cannot be immediately handled, so the system attempts to place the call in a queue of waiting calls. At this point, the caller might hang up; assume the probability the caller hangs up is 0.25, i.e., the caller will remain on the line with probability 0.75. If the caller remains on the line the call is queued; queued calls are handled in first-come-first-serve order. Assume callers do not hang up once they have joined the queue.

Assume the amount of time for an agent to answer an incoming call is uniformly distributed over the interval [300, 700] seconds, independent of the type of agent. Once an agent completes a call, the system will automatically route the next queued call to that agent. If there are no waiting calls the agent becomes idle.

In addition to handling incoming calls, the call center has an automated dialer system that places outbound calls. Every 60 seconds the system simultaneously places some number of outgoing calls. Dialing and determining which calls got through requires 10 seconds. Each call that got through is routed to a sales agent if one is available. Recall service agents cannot handle outgoing calls. Assume the length of each outgoing call is uniformly distributed over [200, 400] seconds. If there are no sales agents available, the outgoing call is dropped and the call is said to be *abandoned*.

A challenge in designing the call system is one does not know how many outbound calls to attempt. Ideally (from the call center's perspective), the number of successful outgoing calls is exactly equal to the number of available sales agents. If the actual number of calls that get through is less than this number, some agents will remain idle, wasting resources. On the other hand, if the number

of successful calls exceeds the number of available sales agents, then the extra successful calls will be abandoned, annoying call recipients (if they're not annoyed already!). Assume that in reality, the probability an outgoing call is successful is $S_{TRUE}$. Of course, the call system does not know the value of $S_{TRUE}$, so cannot use this information to determine how many calls to attempt.

You must develop and evaluate a simple, *adaptive* approach to determine how many outgoing calls to attempt. Do this by trying to estimate $S_{TRUE}$. Specifically, your scheme should compute an estimated probability value $S_{EST}$. When the system places its outgoing calls, it determines the number of idle sales agents, $N_{IDLE}$, and places $N_{IDLE}/S_{EST}$ calls. Note that there is a delay from when the calls are initiated until the calls can be handed off to agents so the number of available sales agents may not be $N_{IDLE}$ when it is time to assign successful calls to agents.

Propose an adaptive scheme for determining $S_{EST}$ and implement it in your simulation. For example, if the system experiences many abandoned calls, this suggests $S_{EST}$ is set too small and should be increased to reduce the number of calls that are made. Conversely, if there are many sales agents left idle, then you may want to reduce $S_{EST}$ to make more calls. You must consider information such as the number of idle or abandoned calls in the recent past, and adjust $S_{EST}$ in accordance. Your scheme could be as simple as only considering the calls placed in the previous round of outgoing calls, or it could consider a recent history spanning several previous rounds.

Your simulation should compute the following metrics:

- Percent of incoming calls where (1) no waiting was required, (2) the caller hung up, and (3) the caller waited in the queue (note these should total to 100%).
- Average time a caller had to wait in the queue; this statistic should only consider calls that were placed in the queue.
- Average number of "successful" outgoing calls per hour (including abandoned calls); percentage of successful outgoing calls that were abandoned.
- Average number of idle service agents and average number of idle sales agents, based on 60 second samples of these two quantities.

Construct a set of runs that vary the arrival rate of incoming calls. Generate graphs showing each of the statistics shown above as the arrival rate is varied and explain your results. From these experiments, select three different arrival rates that will be characterized as "light," "medium," and "heavy" traffic. There should be some, but only a modest amount of queueing occurring for light traffic, a moderate amount of queueing for medium, and substantial queueing under heavy traffic.

Construct and run experiments to evaluate your approach to generating outgoing calls. In particular, start the simulation with $S_{TRUE}$ set to 0.4. Let the simulation run for an hour or so of simulation time to get it into a steady state. Then change $S_{TRUE}$ to 0.6 and let your simulation run to show how it adapts $S_{EST}$. Then some time later change $S_{TRUE}$ back to 0.4. Show on a graph a trace of the values of $S_{TRUE}$ and $S_{TRUE}$ over time. Show how the number of idle agents and abandoned calls change as $S_{EST}$ is changing. Repeat this for light, medium, and heavy traffic.

**Simulation Engine**

Your simulation engine should implement the FEL using the calendar queue data structure (Brown 1988). You must follow the following rules concerning the implementation:

1. Your priority queue implementation must be general, i.e., able to store any number of pending events, limited only by the amount of memory available on the computer executing the program.
2. Use a double precision floating point number to represent simulation time.

3. Each event must be implemented as a C `struct`. Memory for events must be dynamically allocated and released using `malloc()` and `free()`, respectively. Each event must include a timestamp value, and any parameters you deem necessary to characterize the event, e.g., an event type. The simulation engine should be designed so it can be used, as is, for another application using completely different event types and event handlers.

In addition to the FEL, the simulation engine code should include functions for generating random numbers for distribution(s) required by the simulation application. Include the prototypes for these functions as well as other functions, data structures, or variables required by the simulation application in the documentation for the simulation engine. The interface to the simulation engine must be encapsulated in a single C "`.h`" file.

Complete a series of experiments evaluating the performance of your calendar queue as a function of the number of elements in the queue. Write a test program when the queue initially contains N events, then repeatedly (1) remove the smallest timestamped event, and (2) generate and insert a new event with timestamp equal to that of the event just removed from the event list plus an exponentially distributed random value with mean of 1.0. Each cycle through your loop including a remove and insert operation is called a "hold" operation. Perform a large number of hold operations, and compute the average time to perform one hold operation. Plot the time to perform a hold operation as a function of the number of events in the event list, and explain your results. Repeat these experiments for the linear list implementation provided in the sample simulation and show these results on the same graph. Are your results consistent with what you would expect from results reported in Brown's paper? Repeat a set of experiments evaluating the performance of non-static queues, i.e., queues that change their size, as described in the results shown in Figure 5 of the Brown paper. You need only complete experiments for probability values 0.6 and 1.0, as discussed by Brown.

Finally, complete additional experiments to characterize the performance of your priority queue implementation under more realistic test conditions, specifically, the application program developed by your team partner. Complete the experiments described above where the arrival rate is varied. Measure the average insertion and deletion times for the FEL and plot your measurements on a graph. Compare the results with the measurements reported earlier using the synthetic "hold" workload. Is the synthetic data a good predictor of performance when running an actual simulation? Explain why or why not. Document your findings in the project report.

**Implementation Suggestions**

First work with your partner to do a careful review of the sample simulation code to make sure you both understand how it works. Modify the interface between the sample simulation application and simulation engine to fit your needs for the assignment, and document this interface in the `.h` file. Modify the sample code as needed to conform to your interface. This code base can then be used to develop and test the simulation application and engine separately.

To implement the simulation application, we suggest you first get the model for incoming calls to work correctly. First determine what state variables and the types of events you need, and write pseudo-code for the processing of each event before writing your actual code. Execute your pseudo-code "by hand" to convince yourself it works correctly. then implement your program using the (modified) sample simulation engine using the linear event list to test your simulation without having to wait for your partner to complete the new simulation engine. Because you know the average time to process each telephone call and the number of agents, you can easily determine the maximum possible arrival rate the call center can handle; if the arrival rate exceeds this value (or even approaches it), the queue waiting time should increase without bound. Are your

simulation results consistent with this simple analysis? Once you are confident the simulation for incoming calls is working, add the model for outgoing calls. Then integrate your running simulation with your partner's simulation engine, and complete the "production runs" of your simulator to collect the results you need.

For the simulation engine, first develop the utility routines the simulation application will need so those are available to your partner. Then develop and test your priority queue implementation. You will need to implement the resizing operation as well as the insertion and deletion operations. We recommend you first get the insertion and deletion functions to work, then implement the resizing operation.

To generate random numbers from an exponential distribution with mean `U`, create a function `double urand(void)` that returns a random number uniformly distributed over the interval [0,1) (note it cannot return the value 1.0), then define `double randexp()` that returns the value `-U*(log(1.0—urand()))` where `log()` is the C function defined in `<math.h>` to compute a natural logarithm.

Once you are convinced your FEL works, replace the one in the sample simulation with your implementation, and verify the sample simulation produces identical results as the original. Finalize your simulation engine code, and integrate it with your partner's application, and complete the required testing to evaluate the performance of the simulation engine code.

Be sure to allow plenty of time to complete the simulation runs, and write up your results. For example, we recommend aiming to complete the software development by the beginning of the week in which the assignment is due.

**Project Report**

The project report should be a single document developed by your team. It should include:

1. A short introduction section describing the simulation in high level terms.
2. Brief description of the API between the simulation engine and application, noting changes from the sample simulation, if any.
3. Pseudo-code for the simulation application, describing state variables and actions performed by the event handlers.
4. Description of the testing procedure for the simulation application, and evidence the code works correctly.
5. Brief description of the simulation engine, and the FEL implementation.
6. Description of the testing procedure for the simulation engine, and evidence it works correctly.
7. Results of test runs of the *integrated* simulation application and simulation engine, if not reported above, giving evidence the combined simulation is working correctly.
8. Results of the experiments of the simulation application and simulation engine as well as an analysis of the results, and explanations as needed.

**Final Comments**

Turn your report and software in as a single zip file. Your software must be well documented and include comments so the code is easy to understand. You should include a README file with instructions on how to compile and run your program.

The web is an excellent resource to answer specific questions on C. We encourage you to use it, but be careful not to utilize code copied directly from the web. A reminder you must adhere to the Georgia Tech honor code, and the collaboration policy stated in the course syllabus. Specifically,

you are encouraged to discuss the problem and possible solutions with other students (as well as the TA/instructor), however, all code that you turn in must be completely your own work. Disseminating your code to other students (outside your team) is strictly prohibited. Further, downloading code from the web or other sources other than examples provided in class is also prohibited.

**References**

Brown, R. (1988). "Calendar Queues: A Fast O(1) Priority Queue Implementation for the Simulation Event Set Problem." Communications of the ACM **31**(10): 1220-1227.