Simulation project –task 2

Objectives

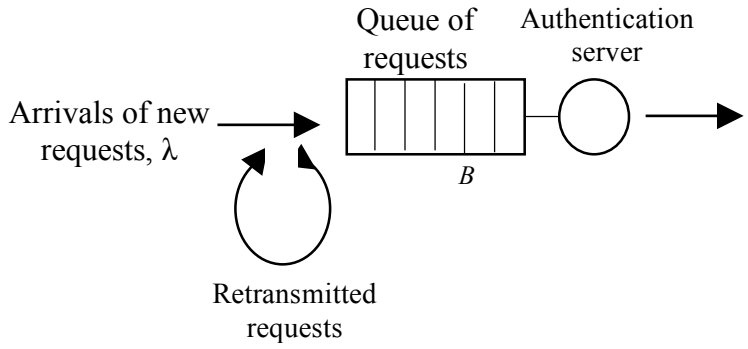
The objective is to simulate the re-certification process of a number of IoT devices with a view to characterizing its perfromance.

You can use any high-level language of your choice. You can develop your program on your personal computer, but it has to run and compile on eos. Programming elegance is not required! What is important is that your program runs correctly. For this, you are *strongly* advised to follow the instructions in this handout!

Remember that any exchange of code with another student is forbidden as it constitutes cheating. We will run Moss at the end of this assignment to verify that there has not been any code sharing.

Project description

We assume that 1,000 IoT devices have to be recertified periodically, and that they do this during a fixed period of 5 hours. The devices send a re-certification request to an authentication server, but not at the same time. Rather, each IoT device waits for a period uniformly distributed in (0, 5] hours, before it transmits its request. In order to avoid overflows of the buffer that is allocated to the certification process which runs on a server, we impose an upper bound B on the queue size (this is the total number of requests waiting to be processed plus the one in service). A request that arrives at the server when there are less than B requests in its queue is allowed to join the queue. Otherwise it is rejected and a message is sent back to the IoT device. In this case the IoT device goes through a random delay d and retransmits its request. (Overflows may occur due to misconfigurations whereby the processing rate of the server and the population of the sensors can be such that the resulting server utilization is very high, thus giving rise to a very large queue which may exceed the buffer space.)



A queueing representation of the model

Due to the large number of IoT devices, it is impossible to simulate each one individually. Instead, we assume that requests arrive in a Poisson manner at the rate of λ=1,000/5x60x60=0.0556/sec. That is, the inter-arrival time of successive new requests is assumed to be exponentially distributed with a mean $1/\lambda$=1/0.0556=17.98 sec. We will refer to these requests as *new request*. A blocked request is retransmitted after a delay *d* and it is referred to as a *retransmitted request.*

Below we describe the implementation of the simulation model.

The simulation structure

The basic idea in a simulation is to track the occurrence of the events in the system that change the state of the system. In our simulation, we have the following events:

1. Arrival of a new request
2. Arrival of a retransmitted request (there may be several pending such arrivals)
3. Completion of a service time

The occurrence of one event may trigger the occurrence of one or more events, as will be seen below.

I. Events and actions

1. Initial conditions at time master clock (MC)=0:
   - Queue empty, server is idle.
   - Predermine the time of the first new arrival, i.e., draw an exponential variate with mean $1/\lambda$, and add it to 0. This is the time of the first new arrival.

2. Events and logic
   1. Arrival of a new request:
      a. Once a new request arrives, determine the next arrival of a new request. That is, draw an exponential variate with mean $1/\lambda$, and add it to the current time indicated by the master clock (MC).

      If the queue size is less than *B*, then the new arrival enters the buffer. If the buffer is empty and the server is idle, then schedule a new service completion.

      b. If the queue size is equal to *B*, then the new arrival is rejected. In this case, the rejected request will be retransmitted after an exponentially distributed delay *d.* Schedule a new event for the arrival of the retransmisted request at time MC+*d*.
   2. Arrival of a retransmitted request:
      a. If the queue size is less than than *B*, the arrival enters the buffer. If the buffer is empty and the server is idle, schedule a new service completion.
      a. If the queue size is equal to *B*, then the arrival is rejected again. In this case, follow 1c.

   2. Service completion:

Once a service is completed, generate the next service completion time if the queue is not empty.

3. Event list
   The event list consists of the events: a) time of arrival of a new request, b) service completion time, and c) time of arrival of a retransmitted request (there may be more than one). That is, it has the following structure:

Next new arrival
Service completion
Arrival of retransmitted request
. . .
Arrival of retransmitted request

   The event list has a variable number of entries, and it should be implemented as a linked list or a fixed array, but be careful so that it does not overflow!

II. Logic of the simulation program

The main logic of the simulation is as follows:

a. Locate the event from the event list that has the smaller clock value *t.* This requires that you search the event list for the smallest clock value so that to identify the next event to execute. Alternatively, you keep the event list sorted in an ascending order according to the event arrival times, and simply select the one at the top of the event list.
b. Advance the simulation to time *t*, i.e., set MCL=*t*, and execute the logic associated with the event, and all other events that may get triggered. Populate the event list accordingly.
c. Check stoppage rule and if you are to continue go back to a.

III. Deliverables

1. Run your simulation using the same parameters as the hand simulation until MC=200, i.e., 1. That is, the stopping rule is the value of the master clock. (We will change this stopping rule in the next task.)
2. Use a pseudo-random number generator to generate random numbers in [0,1]. Use this generator to generate exponential variates for the inter-arrival times and orbiting times. The mean values of these two exponential distributions are the same as the numbers used in the hand simulation. The service time will remain constant equal to the value in the hand simulation, i.e. it will not vary exponentially as the inter-arrival and orbiting times, since the execution time of a request should more or less be constant.

Verify by hand that in both cases the simulation advances correctly from one event to the next one. <u>This is your chance to make sure that your implementation is corect</u> !!

Word of caution: The input values will cause the simulation to become unstable, in the sense that the event list with the orbiting customers may continue to increase. (The queue will never become unstable, because it is finite.) This is done on purpose so that to create different dynamics in the simulation for debugging purposes. We will change these values in task 3.

IV. What to submit

1. Submit your code. To simplify things, submit two different programs, one for task 1 and a separate one for task 2.

   Make sure that the code runs on eos. Also, set up your code so that it would receive input from the terminal. Your output should be saved in a file named output.txt. The input values are
   - the mean inter-arrival time,
   - mean orbiting time,
   - service time,
   - buffer size, and
   - value of the master clock at which time the simulation will be terminated.
   -
2. Submit in a separate file your output from sub-tasks 1 and 2.

V. Grading

60 points if your both version of the code run and compile on eos correctly
40 points if they gives the correct output on test data designed by the TA.