# Computer Assignment 03: Multiple Server Queue with Reneging Customers

## Concepts

* Use of containers
* Passing parameters in waitDelay() method
* Use of interrupt() to implement canceling edge
* Subclassing Entity class for modeling transient entities with additional attributes

## Description

Impatient customers arrive to a multiple-server queue; each customer is only willing to wait a certain amount of time in the queue, after which he or she will “renege.” A reneging customer leaves the queue and never returns to the system. For the model, these “renege times” will be assumed to be independent identically distributed random variables. The Event Graph for the server portion of the model is shown in Figure 1.

The Event Graph in Figure 1 adds reneging by receiving unique customer objects upon arrival to the queue (i.e., at the Arrival event). This customer is added to the end of a container (called ‘q’). The Renege event is then scheduled, with the customer passed as a parameter. When the Renege event occurs, it removes the customer in its argument from the queue and increments the number of reneges (R). Whenever a StartService occurs first, however, the Renege event corresponding to that customer is cancelled.



Figure . Event Graph for Multiple Server Queue with Reneging Customers[[1]](#footnote-1)

In Figure 1, note that the service times are not generated by the Server component as in previous models, but are “carried” along by the Customer objects (described next).

## The Customer Class

The Customer class is a subclass of the Entity class (in the simkit package). Your subclass should add two instance variables, the service time (serviceTime) and the renege time for the customer (renegeTime). The renege time should be passed into the Customer’s constructor as a double and should have a getter method. The (simplified) UML class diagram for the Customer class is shown in Figure 2.



Figure . UML Class Diagram For Customer Class

Note that serviceTime = tS and renegeTime = tR in Figure 1 and that in the code the corresponding getter method should be used to obtain the value.

## The CustomerArrivalProcess Class

Customers will be instantiated by the CustomerArrivalProcess class, which operates in a similar manner as EntityArrivalProcess, from class. It has parameters shown in Table 1 and state show in Table 2.

Table 1. CustomerArrivalProcess Parameters

|  |  |  |
| --- | --- | --- |
| **Parameter** | **Type** | **In Event Graph** |
| interarrivalTimeGenerator | RandomVariate | tA |
| serviceTimeGenerator | RandomVariate | tS |
| renegeTimeGenerator | RandomVariate | tR |

Table 2. CustomerArrivalProcess State Variable

|  |  |  |
| --- | --- | --- |
| **State Variable** | **Type** | **In Event Graph** |
| numberArrivals | int | N |

The Event Graph for CustomerArrivalProcess is shown in Figure 3



Figure . CustomerArrivalProcess Event Graph

In order to produce a double value to pass into the Customer constructor, use the generate() method of RandomVariate. That is:

double renegeTime = renegeTimeGenerator.generate();

double serviceTime = serviceTimeGenerator.generate();

Customer customer = new Customer(serviceTime, renegeTime);

You may either implement the Event Graph for arrivals as exactly in Figure 3 or by sub-classing ArrivalProcess, as in Figure 4.



Figure 4. Subclassing ArrivalProcess

## The ServerWithReneges Class

The ServerWithReneges class processes customers according to the Event Graph in Figure 1. The parameters are shown in Table 1, and the state variables are shown in Table 2.

Table . Parameters for the ServerWithReneges Class

|  |  |  |
| --- | --- | --- |
| **Parameter** | **Type** | **In Event Graph** |
| totalNumberServers | int | k |

Table . State Variables for ServerWithReneges Class

| **State Variable** | **Type** | **Initial Value** | **In Event Graph** |
| --- | --- | --- | --- |
| numberAvailableServers | int | k | S |
| queue | SortedSet<Customer> | empty | q |
| numberReneges | int | 0 | R |
| delayInQueueServed | double | NaN | DS |
| delayInQueueReneged | double | NaN | DR |
| timeInSystem | double | NaN | W |

*Important*: the getter method for queue should return a *copy* of the queue variable as with the EntityServer model.

In the doArrival(Customer) method, the first item is to call stampTime() on the arriving Customer object, as in CustomerServer. The next state transition is to add the customer object to the queue container. Since the queue itself is a state variable, it must fire a PropertyChange event when it changes (i.e. whenever a Customer object is put in or taken out); use the following code snippet to fire the PropertyChange Event:

customer.stampTime();

SortedSet<Customer> oldQueue = getQueue();

queue.add(customer);

firePropertyChange("queue", oldQueue, getQueue ());

The stampTime() call is important since it is the primary sorting key for Entities.

The StartService event has no argument, so the doStartService() method should likewise not either. Inside the method, you will need a reference to the first customer object of the queue:

Customer customer = queue.first();

queue.remove(customer);

Since this changes the state of queue, the firePropertyChange() should be done in a manner similar to that in doArrival(Customer). The customer then should be passed as the last argument in both the interrupt call that cancels Renege and the waitDelay call for the edge that schedules EndService.

Implementing the state variable ‘DS’ (delay in queue for customers served) indicated in Figure 1 should be done in a like manner as in EntityServer (from class). After all states have been changed, invoke the interrupt() method, and finally the waitDelay() method indicated in Figure 1.

All the other do methods in ServerWithReneges can be implemented using concepts that have been covered.

Also, the reset() method should implement the state transitions in the Run event in Figure 1, and doRun() should only fire PropertyChanges (just as in previous computer assignments). As mentioned above, only the time-persistent state variables should fire ProperyChanges in doRun().

### Canceling Edges

Canceling edges are implemented in Simkit by invoking the interrupt() method (defined in SimEntityBase). The form you should use here has signature (String, Object…), where the first argument is the name of the event to be canceled and the second and subsequent arguments are the parameter list corresponding to the event being cancelled. The first event that matches the values of the second parameter (as well as the name of the first parameter) of the interrupt will be removed from the event list. If there is no such event on the event list, then nothing happens.

In this case, the customer who is starting service must have his corresponding Renege event canceled, so the following code is a part of the doStartService() method:

Customer customer = queue.first();

queue.remove(customer);

...

interrupt(“Renege”, customer);

As with a scheduling edge, the call is made in the ‘do’ method corresponding to the event at the tail of the edge. The interrupt() statements should occur after the state transitions but before the waitDelay() statements.

## Execution Class

Write a pure execution (main) class called RunServerWithReneges. The main method should instantiate a CustomerArrivalProcess and a ServerWithReneges and connect the listeners, as shown in Figure 3.



Figure . SimEventListener Diagram

Figure 4 shows the listener diagram for the CustomerArrivalProcess and the ServerWithReneges.

Collecting statistics is basically the same as for EntityServer. You will need one additional one for delayInQueueReneged; the others should be similar to the ones for EntityServer – instances of SimpleStatsTally, SimpleStatsTimeVarying, and CollectionSizeTimeVarying.

### Parameters for Execution

Use the following parameters for executing the model:

* Interarrival times are Exponential(1.5)
* Service times are Gamma(2.5, 1.2)
* Renege times are Uniform(2.0, 6.0)
* Number of servers = 2

## Output

You should use verbose and/or single-step modes to debug your model. When you are satisfied that your model is working correctly, turn off verbose and property dumpers, and perform a run for 100,000.0 time units, producing the following output:

CustomerArrivalProcess.1

renegeTimeGenerator = Uniform (4.000, 6.000)

serviceTimeGenerator = Gamma (2.500, 1.200)

interarrivalTimeGenerator = Exponential (1.500)

ServerWithReneges.2

totalNumberServers = 2

Simulation ended at time 100,000.00

Number Arrivals: 66,653

Number Served: 56,975

Number Reneges: 9,673

Percent Reneges: 14.51%

Avg # in Queue: 1.5222

Avg Utilization: 0.8556

Avg Delay in Queue Served: 1.8630

Avg Delay in Queue Reneged: 4.7613

Avg Time in System: 4.8663

The percent reneges should take into account those customers who have reneged, those customers who have finished service, or are currently in service, but *not* those customers in the queue when the simulation ends.

## Deliverables

Push your code to Gitlab by COB on the due date.

*Important*: Each ‘do’ method *must* have Javadoc comments briefly explaining the event.

1. In the Arrival event, “c.tR” is the renege time for the arriving customer and is to be obtained using the appropriate “getter” method of the Customer class [↑](#footnote-ref-1)