## Computer Assignment 2: The Multiple Server Queue

Objectives

* Gain experience with Simkit
* Implement a component-based model
* Re-use of ArrivalProcess class
* Use of Priority to set scheduling priority
* Communicating between objects with SimEventListener
* Communicate state changes by firing PropertyChangeEvents
* Use SimpleStatsTimeVarying to collect time-varying means

Description

In this assignment you will create a model of the Multiple Server queue (G/G/k) in Simkit using a component design that was described in class and will be repeated below. Your model will re-use the ArrivalProcess class from the first computer assignment as one component; the server functionality will be added by the SimpleServer component, which you will implement in this assignment. You should not have to change any of the code in (a correctly implemented) ArrivalProcess[[1]](#footnote-1). This assignment consists of writing two classes: SimpleServer that will implement the server portion of the queue and a pure execution class, RunSimpleServer, that will (ultimately) run the model for 100,000 time units and collect statistics for a single replication.

Writing the SimpleServer Class

The Event Graph to create the SimpleServer portion of the model is shown in Figure 1. Note that the Event Graph in Figure 1 is not a complete model and cannot run by itself, but is an Event Graph *component*. Specifically, the Run event does not schedule any other events, so there needs to be at least one other component in the model to schedule the first event. The SimpleServer component will work by “listening” to another component for the Arrival event. When the Arrival event is “heard,” the state transition and scheduling depicted in Figure 1 will be triggered. The connection between the Arrival event of the ArrivalProcess component and the Arrival event of the SimpleServer component will be implemented using the addSimEventListener method, which will be described below.

Note that two of the scheduling edges in Figure 1 have a high priority. This is implemented by adding Priority.HIGH as a third argument to the corresponding waitDelay() call, as will be shown below.[[2]](#footnote-2)



Figure 1. SimpleServer Event Graph

SimpleServer Event Graph

Define a class in the mv3302 package called SimpleServer extending SimEntityBase. Define the instance variables for your parameters and state variables as shown in Table 1. As with the ArrivalProcess, your state variables should be declared protected and have getters but no setters, whereas the parameters are private and have both setters and getters. The primary constructor signature should be (int, RandomVariate) and it should simply call the two corresponding setter methods. You should also write a constructor with zero arguments.

|  |  |
| --- | --- |
| **State Variables** | **Parameters** |
| numberInQueue (int) | totalNumberServers (int) |
| numberAvailableServers (int) | serviceTimeGenerator (RandomVariate) |
| numberServed (int) |  |

Table 1. Parameters and State Variables for SimpleServer Class

The setter method for totalNumberServers should throw an IllegalArgumentException if it is passed a value that is zero or less. Start with the generated setter method, and then modify as shown below. Be sure to include the Javadoc comment.

/\*\*

\* @throws IllegalArgumentException if totalNumberServers <= 0

\* @param totalNumberServers the totalNumberServers

\*/

public void setTotalNumberServers(int totalNumberServers) {

if (totalNumberServers <= 0) {

throw new IllegalArgumentException("totalNumberServers must be > 0: " +

totalNumberServers);

}

this.totalNumberServers = totalNumberServers;

}

Next, write the “do” methods. As with the ArrivalProcess, each event in Figure 1 will correspond to a method with “do” prefixed (“doArrival()”, “doStartService()”, and “doEndService()”). You should also write reset() and doRun() methods, which will now be discussed.

The reset() method is necessary in this class for setting the initial values of numberInQueue (to 0) and numberAvailableServers (to totalNumberServers). Remember that whenever the value of a state variable changes, a PropertyChange event should be fired, as you did in the ArrivalProcess class and in the doArrival() method above. However, the initial values of state variables are set in reset() but the firePropertyChange calls are in the doRun() method. Thus, your reset() method should look like this:

public void reset() {

super.reset();

numberInQueue = 0;

numberAvailableServers = totalNumberServers

numberServed = 0;

}

The first call in reset() should always be to super.reset().

In general, the doRun() method should fire property changes for time-varying state variables, which in this case is all three. The doRun() method therefore looks like this:

public void doRun() {

firePropertyChange("numberInQueue", getNumberInQueue());

firePropertyChange("numberAvailableServers", getNumberAvailableServers());

firePropertyChange("numberServed", getNumberServed());

}

Your doArrival() method should look like this (yours should be commented, of course):

public void doArrival() {

int oldNumberInQueue = getNumberInQueue();

numberInQueue = numberInQueue + 1;

firePropertyChange("numberInQueue", oldNumberInQueue, getNumberInQueue());

if (getNumberAvailableServers() > 0) {

waitDelay("StartService", 0.0, Priority.HIGH);

}

}

Note that the boolean edge condition is implemented by wrapping the corresponding waitDelay() statement for the edge in an ‘if’ test. Write the other “do” methods in a similar manner, making sure to write a call to waitDelay() for each scheduling edge in Figure 1. The service time delay from the StartService to the EndService event should be implemented in a similar manner as the delay in the ArrivalProcess component.

If the Event Graph calls for it, the Run event may have scheduling edges as well (as with the ArrivalProcess component from computer assignment 1), so the doRun() method in that case would contain calls to waitDelay() for each one. For this component, however, there aren’t any. Therefore, the doRun() method in this case consists only of the above calls to firePropertyChange();

SimEventListener

Since the Run event in Figure 1 does not schedule any other events, there are no calls to waitDelay() in doRun(). Therefore, if there are no other SimEntities that are scheduling events in a model, nothing will happen after the Run event. An instance of a SimpleServer functions by “listening” to another SimEntity that schedules the Arrival event. It so happens that you created such a SimEntity – an instance of the ArrivalProcess class that was developed in class. So you will instantiate an ArrivalProcess and SimpleServer, and then instruct the SimpleServer instance to listen to the ArrivalProcess instance. This is depicted in Figure 2.

Listeners

Figure 2. SimEventListener Diagram

The connecter in Figure 2 indicates that an instance of SimpleServer listens to the events of the ArrivalProcess instance; the connecter can be thought of like a stethescope. The Simkit code corresponding to Figure 2 is written as follows:

arrivalProcess.addSimEventListener(simpleServer);

where arrivalProcess is the ArrivalProcess instance and simpleServer is the SimpleServer instance.

Running the Model

Write a pure execution class (i.e. having only a main method) called RunSimpleServer. The main method should:

1. Instantiate an ArrivalProcess (call it “arrivalProcess”) with Uniform(0.9, 2.2) interarrival times:

RandomVariate interarrivalTime =

RandomVariateFactory.getInstance("Uniform", 0.9, 2.2);

ArrivalProcess arrivalProcess = new ArrivalProcess(interarrivalTime);

Use a gamma distribution for the service times. Recall that the gamma distribution has two parameters, and , and that the mean and variance for a gamma random variable are and  respectively. The gamma random variable generator in Simkit takes  and  as its parameters. So, in your main method obtain a RandomVariate instance with a Gamma (1.7, 1.8) distribution as follws:.

String rvName = “Gamma”;

double alpha = 1.7;

double beta = 1.8

RandomVariate serviceTime =

RandomVariateFactory.getInstance( rvName, alpha, beta );

1. Instantiate a SimpleServer:

SimpleServer simpleServer = new SimpleServer(2, serviceTime);

Note that for this example there will be 2 servers.

1. Print the arrivalProcess and simpleServer instances to ensure that the correct values have been passed to their respective parameters.
2. Add simpleServer as a SimEventListener to arrival as follows (in main):

arrivalProcess.addSimEventListener(simpleServer);

1. Instantiate a SimplePropertyDumper and have it listen to the SimpleServer instance’s property changes, in a similar manner to when the ArrivalProcess was run in class.
2. Finally, run the model in verbose mode for 6.0 time units, just as a test:

Schedule.stopAtTime(6.0);

Schedule.setEventSourceVerbose(true);

Schedule.setVerbose(true);

Schedule.reset();

Schedule.startSimulation();

Compile and run. Your output should look something like this:

ArrivalProcess.1

interarrivalTimeGenerator = Uniform (0.900, 2.200)

SimpleServer.2

serviceTimeGenerator = Gamma (1.700, 1.800)

totalNumberServers = 2

\*\* Event List 0 -- Starting Simulation \*\*

0.000 Run <ArrivalProcess.1>

0.000 Run <SimpleServer.2>

6.000 Stop <Stop.3>

\*\* End of Event List -- Starting Simulation \*\*

numberArrivals: 0

Time: 0.0000 CurrentEvent: Run [1]

\*\* Event List 0 -- \*\*

0.000 Run <SimpleServer.2>

1.963 Arrival <ArrivalProcess.1>

6.000 Stop <Stop.3>

\*\* End of Event List -- \*\*

numberInQueue: 0

numberAvailableServers: 2

numberServed: 0

Time: 0.0000 CurrentEvent: Run [2]

\*\* Event List 0 -- \*\*

1.963 Arrival <ArrivalProcess.1>

6.000 Stop <Stop.3>

\*\* End of Event List -- \*\*

numberArrivals: 0 => 1

numberInQueue: 0 => 1

Time: 1.9625 CurrentEvent: Arrival [1]

\*\* Event List 0 -- \*\*

1.963 StartService <SimpleServer.2>

4.161 Arrival <ArrivalProcess.1>

6.000 Stop <Stop.3>

\*\* End of Event List -- \*\*

numberInQueue: 1 => 0

numberAvailableServers: 2 => 1

Time: 1.9625 CurrentEvent: StartService [1]

\*\* Event List 0 -- \*\*

4.161 Arrival <ArrivalProcess.1>

5.105 EndService <SimpleServer.2>

6.000 Stop <Stop.3>

\*\* End of Event List -- \*\*

numberArrivals: 1 => 2

numberInQueue: 0 => 1

Time: 4.1613 CurrentEvent: Arrival [2]

\*\* Event List 0 -- \*\*

4.161 StartService <SimpleServer.2>

5.105 EndService <SimpleServer.2>

5.107 Arrival <ArrivalProcess.1>

6.000 Stop <Stop.3>

\*\* End of Event List -- \*\*

numberInQueue: 1 => 0

numberAvailableServers: 1 => 0

Time: 4.1613 CurrentEvent: StartService [2]

\*\* Event List 0 -- \*\*

4.695 EndService <SimpleServer.2>

5.105 EndService <SimpleServer.2>

5.107 Arrival <ArrivalProcess.1>

6.000 Stop <Stop.3>

\*\* End of Event List -- \*\*

numberAvailableServers: 0 => 1

numberServed: 0 => 1

Time: 4.6946 CurrentEvent: EndService [1]

\*\* Event List 0 -- \*\*

5.105 EndService <SimpleServer.2>

5.107 Arrival <ArrivalProcess.1>

6.000 Stop <Stop.3>

\*\* End of Event List -- \*\*

numberAvailableServers: 1 => 2

numberServed: 1 => 2

Time: 5.1055 CurrentEvent: EndService [2]

\*\* Event List 0 -- \*\*

5.107 Arrival <ArrivalProcess.1>

6.000 Stop <Stop.3>

\*\* End of Event List -- \*\*

numberArrivals: 2 => 3

numberInQueue: 0 => 1

Time: 5.1073 CurrentEvent: Arrival [3]

\*\* Event List 0 -- \*\*

5.107 StartService <SimpleServer.2>

6.000 Stop <Stop.3>

6.183 Arrival <ArrivalProcess.1>

\*\* End of Event List -- \*\*

numberInQueue: 1 => 0

numberAvailableServers: 2 => 1

Time: 5.1073 CurrentEvent: StartService [3]

\*\* Event List 0 -- \*\*

6.000 Stop <Stop.3>

6.183 Arrival <ArrivalProcess.1>

7.757 EndService <SimpleServer.2>

\*\* End of Event List -- \*\*

Time: 6.0000 CurrentEvent: Stop [1]

\*\* Event List 0 -- \*\*

6.183 Arrival <ArrivalProcess.1>

7.757 EndService <SimpleServer.2>

\*\* End of Event List -- \*\*

Simulation ended at time 6.000

There have been 3 arrivals

There have been 2 customers served

Note that the simpleServer object has had its Arrival event triggered by the ArrivalProcess object’s Arrival event. If you are not getting all the events listed above, check that you have set up the SimEventListener correctly and have spelled the ‘do’ methods correctly.

Collecting Statistics

Now you are going to run the model for a longer period of simulated time and collects some statistics. Simkit provides a class in the simkit.stat package called SimpleStatsTimeVarying that can estimate a time-varying mean from values fired as PropertyChanges.

Modify RunSimpleServer to run your model and collect statistics. After instantiating the ArrivalProcess and SimpleServer objects, add code to instantiate an object of type SimpleStatsTimeVarying as follows:

SimpleStatsTimeVarying numberInQueueStat

= new SimpleStatsTimeVarying("numberInQueue");

The String passed to the constructor above, “numberInQueue”, has the same name (case-sensitive) as the property that was fired in SimpleServer. Instantiate another one for numberAvailableServers. These should be done after the SimpleServer and ArrivalProcess objects are instantiated but before the Simulation methods are invoked. Finally, connect it to the simpleServer as a PropertyChangeListener as follows:

simpleServer.addPropertyChangeListener(numberInQueueStat);

Create a SimpleStatsTimeVarying and have it listen to simpleServer for changes to the numberAvailableServers state variable in a similar manner.

When this compiles and works for short runs, set it to stop at time 100000.0, set the verbose mode to false, remove the SimplePropertyDumper (or comment it out), and output the mean values after the run. Use the getMean() method of SimpleStatsTimeVarying: after the simulation has run (i.e. after startSimulation() has returned), numberInQueueStat.getMean() gives the average number in queue for that run.

The average utilization is defined to be 1.0 - (avg # available servers) / total number of servers. The code to echo back the parameters of the model should be written in main before Schedule.startSimulation(); the code to write the output statistics should also be in main but come after Schedule.startSimulation():

System.out.printf("Simulation ended at time %,.3f%n", Schedule.getSimTime());

System.out.printf("%nThere have been %d arrivals%n", arrival.getNumberArrivals());

System.out.printf("There have been %d customers served%n",

server.getNumberServed());

System.out.printf("Average number in queue\t%.3f%n",

numberInQueueStat.getMean());

System.out.printf("Average utilization\t%.3f%n", 1.0 –

numberAvailableServers.getMean() / server.getNumberServers());

Your final output should look like this:

ArrivalProcess.1

interarrivalTimeGenerator = Uniform (0.900, 2.200)

SimpleServer.2

totalNumberServers = 2

serviceTimeGenerator = Gamma (1.700, 1.800)

Simulation ended at time 100,000.000

There have been 64,475 arrivals

There have been 64,472 customers served

Average number in queue 15.965

Average utilization 0.982

To get this output, use getter methods from the SimpleServer and ArrivalProcess as well as getMean() from SimpleStatsTimeVarying. The end time should be obtained using Schedule.getSimTime(). Use java.textDecimalFormat to format the numbers with only four digits to the right of the decimal point, as indicated above.

Deliverables

Push your final code in your MV3302 project to Gilab by COB of the due date.

**Important Note**: Your SimpleServer class *must* have all its ‘do’ methods and reset() commented in Javadoc format. Each comment must describe what the method does – its state transitions and Event(s) that are scheduled (if any).

Frequently Asked Questions

### What does addSimEventListener do?

After the listenee executes an event from the Event List, it passes that event to the listener. If the listener has an event that matches, then that event is executed. In this program, the SimpleServer instance has its Arrival event triggered by the ArrivalProcess’s Arrival event.

### What’s with all this firePropertyChange stuff?

Simkit can exploit the JavaBeans property listener pattern by having only those objects who are “interested” in a given property registering that interest and receiving a PropertyChangeEvent when the property changes value. The firePropertyChange() method dispatches a PropertyChangeEvent to all registered listeners for the object with the property. Although this is a little more work now, the property change listener pattern makes things much easier down the road.

### I got ArrivalProcess.2 and SimpleServer.1 in my printout.

There is no problem with this - it only means you instantiated your SimpleServer first and ArrivalProcess second. Your results should be otherwise identical.

### Please give more information about what is required for comments.

Setters and getters do not need to be commented, unless they do something other than pass the value (see the comments suggested for setTotalNumberServers above). The comment for a ‘do’ method should describe the state transitions and each scheduling edge. For example, an acceptable comment for doEndService() looks like this:

/\*\*

\* Increment numberAvailableServers <br>

\*

\* If numberInQueue &gt; 0 (there are customers in queue), schedule

\* StartService with delay of 0.0 and HIGH priority

\*/

public void doEndService() {

. . .

Non-Javadoc comments are useful of course, but are not a substitute for Javadoc, since they will not be included when the Javadocs are generated.

1. Be sure to make any necessary corrections to your ArrivalProcess before doing this assignment! [↑](#footnote-ref-1)
2. See Section 4.5 of *Discrete Event Simulation Modeling* for a description of priorities on scheduling edges. [↑](#footnote-ref-2)