The Bank: an example of a SimPy Simulation

G A Vignaux

2004 Dec 5

SimPy Version: 1.4 or later

Table Of Contents

The Bank tutorial demonstrates the use of *SimPy* in developing and running a simple simulation of a practical problem, a multi-server bank.

Introduction

This tutorial works through the development of a simulation of a bank using the Python-based discrete-event simulation system, SimPy.

SimPy provides *Processes* to model active components such as messages, customers, trucks, and planes. It provides *Resources* to model congestion points such as queues. *Monitors* are a mechanism for making observations of data such as delay times and numbers of entities in the system. Random number generation is supplied by the standard Python random package. SimPy is compatible with Python version 2.2 onwards[1]. *SimPy* itself can be obtained from: http://sourceforge.net/projects/simpy.

Before attempting to use SimPy you should have some familiarity with the the Python language. In particular, you should be able to use and define classes of objects. Python is free and available on most machine types. You can find out more about it and download it from the Python web site. This document assumes *Python* 2.2 or later.

This tutorial should be read with the SimPy Manual or Cheatsheet at your side for reference.

The programs are available without line numbers and ready to go, in directory *bankprograms*. Some have trace statements for demonstration purposes, others produce graphical output to the screen. Let me encourage you to run them and modify them for yourself

Our Bank

We are going to model a simple bank with a number of tellers and customers arriving at random. I should remind you that all simulations need to answer a question; in this case we are to investigate how increasing the number of tellers reduces the queueing time for customers.

We will develop the model step-by-step, starting out very simply.

A Customer

We first model a single customer who arrives at the bank for a visit, looks around for a time and then leaves. There are no bank counters in this model. First we will assume his arrival time and time in the bank are known. Then we will allow the customer to arrive at a random time.

The single non-random Customer

We define a Customer class which will derive from the *SimPy* Process class. Examine the following listing which, except for the line numbers I have added, is a complete runnable Python script. The customer visits the bank at simulation time 5.0 and leaves after 10.0. We will take time units to be minutes

```
 1 #!/usr/bin/env python
 2 """ bank01: The single non-random Customer"""
 3 from __future__ import generators
 4 from SimPy.Simulation  import *
 5
 6 class Customer(Process):
 7     """ Customer arrives, looks around and leaves """
 8
 9     def visit(self,timeInBank):
10         print "%7.4f %s: Here I am"%(now(),self.name)
```

[1] If you use Python version 2.2, you must place the import statement: `from __future __ import generators` at the top of all *SimPy* scripts. This is not needed for Python versions 2.3 on.

```
11          yield hold,self,timeInBank
12          print "%7.4f %s: I must leave"%(now(),self.name)
13
14 def model():
15      initialize()
16      c=Customer(name="Klaus")
17      activate(c,c.visit(timeInBank=10.0),delay=5.0)
18      simulate(until=100.0)
19
20 model()
```

Line 1 is a Python comment indicating to the Unix shell which Python interpreter to use (it is ignored by other operating systems) Line 2 is a normal Python documentation string.

Line 3 is needed only for those still using Python 2.2. It imports generators (from the __future__ as version 2.2 does not have generators built-in). This line will be left out of future programs in this tutorial.

Line 4 imports the the SimPy simulation code.

Now examine the *Customer* class definition, lines 6-12. It defines our customer and has the required generator method (*visit*) (line 9), called in SimPy jargon a Process Execution Method (PEM).

It is always possible to define an *__init__()* method if you need to give the Customer any attributes. An *__init__* method must first call *Process.__init__(self)* and can then initialize any instance variables needed.

The *visit* action method, lines 9-12, executes the customer's activities. When he arrives (it will turn out to be a 'he' in this model), he will print out the simulation time, *now()*, and his name (line 10). The function *now()* can be used at any time in the simulation to find the current simulation time. The name will be set when the customer is created in the *main()* routine.

He then stays in the bank for a simulation period *timeInBank* (line 11). This is achieved in the *yield hold,self,timeInBank* statement. This is the first of the special simulation commands that *SimPy* offers. The *yield* statements demonstrates that the customer's *visit* method is a Python generator.

After a simulation time of *timeInBank*, the program's execution returns to the line after the *yield* statement, line 15. Here he again print out the current simulation time and his name. This completes the declaration of the *Customer* class.

Lines 14-20 declare the model routine and then call it. The name of this function could be anything, of course, (for example *main()*, or *bank()*). Indeed, the code could even have been written in-line rather than embedded in a function but when we come to carry out series of experiments we will find it is better to structure the model this way.

Line 15 calls *initialize()* which sets up the simulation system ready to receive *activate* calls. Then, in line 16, we create a customer with name *Klaus*. All SimPy Processes have a *name* attribute. We *activate Klaus* in line 17 specifying the active object(c) to be activated, the call of the action routine (*c.visit(timeInBank=10.0)*) and the time it is to be activated (with, here, a *after=5.0*). This will activate *klaus* after a delay from the current time, in this case at the start of the simulation, 0.0, of 5.0 minutes. The call of an action routine such as *c.visit* can specify the values of arguments, here the *timeInBank*.

Finally the call of *simulate(until=100.0* in line 18 will start the simulation. It will run until the simulation time is 100.0 unless stopped beforehand either by the command *stopSimulation()* or by running out of events to execute (as will happen here).

So far we have been declaring a class, its methods, and one routine. The call of *model()* in line 20 starts the script running. The trace printed out by the *print* commands shows the result. The program finishes at simulation time 15.0 because there are no further events to be executed. At the end of the *visit* routine, the customer has no more actions and no other objects or customers are active.

The output from the program is

```
5.0000 Klaus: Here I am
```

```
15.0000 Klaus: I must leave
```

The single Random Customer

Now we extend the model to allow our customer to arrive at a random simulated time. We modify the previous model:

```
 1 #!/usr/bin/env python
 2 """ bank05: The single Random Customer"""
 3 from SimPy.Simulation  import *
 4 from random import expovariate, seed
 5
 6 class Customer(Process):
 7     """ Customer arrives at a random time,
 8         looks around  and then leaves
 9     """
10
11     def visit(self,timeInBank=0):
12         print "%7.4f %s: Here I am"%(now(),self.name)
13         yield hold,self,timeInBank
14         print "%7.4f %s: I must leave"%(now(),self.name)
15
16 def model():
17     seed(99999)
18     initialize()
19     c=Customer(name="Klaus")
20     t = expovariate(1.0/5.0)
21     activate(c,c.visit(timeInBank=10.0),delay=t)
22     simulate(until=100.0)
23
24 model()
```

The change occurs in lines 4, 17, 20, and 21 of *model()*. In line 4 we import from the standard Python *random* module. We need *expovariate* to generate the random time of arrival and *seed* to initialise the random number stream. Recall that this gives us control over the random numbers and it will be needed when use the seed as an argument to the *model* routine. In line 17 we provide a seed of 99999. We generate an exponential random variate in line 20 and the random sample, t, is used in Line 21 as the delay argument to the *activate* call.

The result is shown below where we see the customer now arrives at time 10.5809. Changing the seed value would change that time.

```
10.5809 Klaus: Here I am
20.5809 Klaus: I must leave
```

More Customers

Our simulation does little so far. Eventually, we need multiple customers to arrive at random and be served for times that can also be random. Let us next consider several customers. We first go back to the simple deterministic model and add more Customers.

```
 1 #!/usr/bin/env python
 2 """ bank02: More Customers """
 3 from SimPy.Simulation  import *
 4
 5 class Customer(Process):
 6     """ Customer arrives, looks around and leaves """
 7
 8     def visit(self,timeInBank=0):
 9         print "%7.4f %s: Here I am"%(now(),self.name)
10         yield hold,self,timeInBank
11         print "%7.4f %s: I must leave"%(now(),self.name)
12
13 def model():
14     initialize()
15     c1=Customer(name="Klaus")
16     activate(c1,c1.visit(timeInBank=10.0),delay=5.0)
17     c2=Customer(name="Tony")
18     activate(c2,c2.visit(timeInBank=8.0),delay=2.0)
19     c3=Customer(name="Evelyn")
20     activate(c3,c3.visit(timeInBank=20.0),delay=12.0)
21     simulate(until=400.0)
22
23 model()
```

The program is almost as easy as The single non-random Customer. The only change is in lines 15-20 where we create, name, and activate three customers in *model()* and a change in the argument of *simulate(until=100)*. Each of the customers stays for a different *timeinbank*. Note that we only need one definition of the *Customer* class and create several objects of that class which all act quite independently in this model.

Note carefully that customer *Tony* is created second but as it is activated at simulation time 2.0 he will start before Customer *Klaus*.

The trace produced by the program is shown below. Again the simulation finishes before the *till=100.0* of the *simulate* call.

```
 2.0000 Tony: Here I am
 5.0000 Klaus: Here I am
10.0000 Tony: I must leave
12.0000 Evelyn: Here I am
15.0000 Klaus: I must leave
32.0000 Evelyn: I must leave
```

Many non-random Customers

Another change will allow us to have multiple customers. As it is tedious to give a specially chosen name to each customer, we will instead call them *Customer00, Customer01, ...* and use a separate *Source* class to create and activate them.

```
 1 #!/usr/bin/env python
 2 """ bank03: Many non-random Customers """
 3 from SimPy.Simulation  import *
 4
```

```
 5 class Source(Process):
 6     """ Source generates customers regularly"""
 7
 8     def generate(self,number,interval):
 9         for i in range(number):
10             c = Customer(name = "Customer%02d"%(i,))
11             activate(c,c.visit(timeInBank=12.0))
12             yield hold,self,interval
13
14 class Customer(Process):
15     """ Customer arrives, looks round and leaves """
16
17     def visit(self,timeInBank=0):
18         print "%7.4f %s: Here I am"%(now(),self.name)
19         yield hold,self,timeInBank
20         print "%7.4f %s: I must leave"%(now(),self.name)
21
22 def model():
23     initialize()
24     source=Source()
25     activate(source,source.generate(5,10.0),0.0)
26     simulate(until=400.0)
27
28 model()
29
```

The listing shows the new program. Lines 5-12 show the *Source* class. The PEM, here called *generate*, is defined in lines 8-12. This method has a couple of arguments, the *number* of customers to be generated and *interval*, the time between arrivals. It consists of a loop that creates a sequence of numbered *Customers* from 0 to *number-1*. We construct a name and create each one in line 10. Each is then activated at the current simulation time (the final argument of the *activate* statement is missing so that *now()* is used as the time). We also specify how long the customer is to stay in the bank. To keep it simple, all customers will stay exactly 12 minutes. After a new customer is activated, the *Source* holds for a fixed time (*yield hold,self, interval*) before creating the next one.

The *Source* is created in line 24 and activated at line 25 where the number of customers is set to 5 and the interval to 10.0. Once it starts, at time 0.0 it creates customers at intervals and each customer then operates independently of others:

```
 0.0000 Customer00: Here I am
10.0000 Customer01: Here I am
12.0000 Customer00: I must leave
20.0000 Customer02: Here I am
22.0000 Customer01: I must leave
30.0000 Customer03: Here I am
32.0000 Customer02: I must leave
40.0000 Customer04: Here I am
42.0000 Customer03: I must leave
52.0000 Customer04: I must leave
```

Many Random Customers

We extend this model to allow our customers to arrive at random. In simulation this is usually interpreted as meaning that the times between customer arrivals are distributed as exponential random variates. Thus there is little change in our program:

```
 1 #!/usr/bin/env python
 2 """ bank06: Many Random Customers """
 3 from SimPy.Simulation  import *
 4 from random import expovariate,seed
 5
 6 class Source(Process):
 7     """ Source generates customers randomly"""
 8
 9     def generate(self,number,interval):
10         for i in range(number):
11             c = Customer(name = "Customer%02d"%(i,))
12             activate(c,c.visit(timeInBank=12.0))
13             t = expovariate(1.0/interval)
14             yield hold,self,t
15
16 class Customer(Process):
17     """ Customer arrives, looks round and leaves """
18
19     def visit(self,timeInBank=0):
20         print "%7.4f %s: Here I am"%(now(),self.name)
21         yield hold,self,timeInBank
22         print "%7.4f %s: I must leave"%(now(),self.name)
23
24 def model():
25     seed(99999)
26     initialize()
27     s=Source(name='Source')
28     activate(s,s.generate(5,10.0),0.0)
29     simulate(until=400.0)
30
31 model()
```

The exponential random variate is generated in line 13 with *interval* as the mean of the distribution and used in line 14. Note that this parameter is not exactly intuitive. The Python *expovariate* method uses the *rate* of arrivals as the parameter not the average interval. This is in line with the way that queueing theory handles the distribution but experience has shown that this is a common source of error in using SimPy. The exponential delay between two arrivals gives pseudo-random arrivals, as specified. In this model the first customer arrives at time 0.

The *seed* method is called to set up the random numbers in the *model* routine (line 25). The next step would be to use the seed value as an argument of the *model* routine. It is possible to leave this call out but the important point is that if we wish to do serious comparisons of systems, we need control over the random variates and hence control over the seeds. Thus we must be able to run identical models with different seeds or different models with identical seeds. This requires us to be able to provide the seeds as control parameters of the run. Here, of course it is just assigned in line 25 but it is clear it could have been read in or provided in a GUI form.

```
0.0000 Customer00: Here I am
```

```
12.0000 Customer00: I must leave
21.1618 Customer01: Here I am
32.8968 Customer02: Here I am
33.1618 Customer01: I must leave
33.3790 Customer03: Here I am
36.3979 Customer04: Here I am
44.8968 Customer02: I must leave
45.3790 Customer03: I must leave
48.3979 Customer04: I must leave
```

A Counter

Now we extend the bank, and the activities of the customers by installing a counter with a clerk where banking takes place. So far, it has been more like an art gallery, the customers entering, looking around, and leaving. We need a object to represent the counter and *SimPy* provides a *Resource* class for this purpose. The actions of a *Resource* are simple: customers *request* a clerk, if she is free he gets service but others are blocked until the clerk becomes free again. This happens when the customer completes service and *releases* the clerk. If a customer requests service and the clerk is busy, the customer joins a queue until it is their turn to be served.

One Counter

The next model is a development of the previous one with random arrivals but they each need to use a single counter for a fixed time and may have to queue.

```
 1 #!/usr/bin/env python
 2 """ bank07: One Counter"""
 3 from SimPy.Simulation  import *
 4 from random import expovariate, seed
 5
 6 class Source(Process):
 7     """ Source generates customers randomly"""
 8
 9     def generate(self,number,interval):
10         for i in range(number):
11             c = Customer(name = "Customer%02d"%(i,))
12             activate(c,c.visit(timeInBank=12.0))
13             t = expovariate(1.0/interval)
14             yield hold,self,t
15
16 class Customer(Process):
17     """ Customer arrives, is served and  leaves """
18
19     def visit(self,timeInBank=0):
20         arrive=now()
21         print "%7.4f %s: Here I am     "%(now(),self.name)
22
23         yield request,self,counter
24         wait=now()-arrive
25         print "%7.4f %s: Waited %6.3f"%(now(),self.name,wait)
26         yield hold,self,timeInBank
```

```
27          yield release,self,counter
28
29          print "%7.4f %s: Finished      "%(now(),self.name)
30
31 def model():
32     global counter
33     seed(99999)
34     counter = Resource(name="Karen")
35     initialize()
36     source=Source('Source')
37     activate(source,source.generate(5,10.0),0.0)
38     simulate(until=400.0)
39
40 model()
```

The *counter* is created as a *Resource* in lines 32-34. I have chosen to make this a global object so it can be referred to by the *Customer* class. An alternative would have been to create it globally outside *model* or to carry it as an argument of the *Customer* class.

The actions involving the *counter* are the *yield* statements in lines 23 and 27 where we *request* it and then, later (line 27) *release* it. In between there is a *yield* statement corresponding to the time the counter is being used. This pattern of (yield request; yield hold; yield release) is the way that Resources are used.

To show the effect of the counter on the activities of the customers, I have added line 20 that records when the customer arrived and line 24 that records the time between arrival and getting the counter. Line 24 is *after* the *yield request* command and will be reached only when the request is satisfied. It is *before* the *yield hold* that corresponds to the service-time. Queueing may have taken place and *wait* will record how long the customer waited. This technique of saving the arrival time in a variable to record the time taken in systems is common in simulations. So the *print* statement also prints out how long the customer waited and we see that, except for the first customer (we still only have five customers, see line 37) all the customers have to wait. Here is the output:

```
 0.0000 Customer00: Here I am
 0.0000 Customer00: Waited  0.000
12.0000 Customer00: Finished
21.1618 Customer01: Here I am
21.1618 Customer01: Waited  0.000
32.8968 Customer02: Here I am
33.1618 Customer01: Finished
33.1618 Customer02: Waited  0.265
33.3790 Customer03: Here I am
36.3979 Customer04: Here I am
45.1618 Customer02: Finished
45.1618 Customer03: Waited 11.783
57.1618 Customer03: Finished
57.1618 Customer04: Waited 20.764
69.1618 Customer04: Finished
```

A counter with a random service time

We retain the single counter but bring in another source of variability, making the customer service time at the counter a random variable in addition to the inter-arrival time. As is traditional in the study of queues we first assume an exponential service time with a mean of *timeInBank*.

```
 1 #!/usr/bin/env python
 2 """ bank08: A counter with a random service time"""
 3 from SimPy.Simulation  import *
 4 from random import expovariate, seed
 5
 6 class Source(Process):
 7     """ Source generates customers randomly"""
 8
 9     def generate(self,number,interval):
10         for i in range(number):
11             c = Customer(name = "Customer%02d"%(i,))
12             activate(c,c.visit(timeInBank=12.0))
13             t = expovariate(1.0/interval)
14             yield hold,self,t
15
16 class Customer(Process):
17     """ Customer arrives, is served and leaves """
18
19     def visit(self,timeInBank=0):
20         arrive=now()
21         print "%7.4f %s: Here I am     "%(now(),self.name)
22         yield request,self,counter
23         wait=now()-arrive
24         print "%7.4f %s: Waited %6.3f"%(now(),self.name,wait)
25         tib = expovariate(1.0/timeInBank)
26         yield hold,self,tib
27         yield release,self,counter
28         print "%7.4f %s: Finished      "%(now(),self.name)
29
30 def model(theseed):
31     global counter
32     seed(theseed)
33     counter = Resource(name="Karen")
34
35     initialize()
36     source = Source('Source')
37     activate(source,source.generate(5,interval=10.0),0.0)
38     simulate(until=400.0)
39
40 model(theseed=12345)
41
```

The argument, *theseed* is set up as an argument of the *model* function in line 30. Only one common random number stream will be used for both arrivals and service times. The seed is set we call the *model* in line 40.

The new random variables are generated in line 25 and used in line 26 of the *visit()* method of *Customer*.

```
 0.0000 Customer00: Here I am
 0.0000 Customer00: Waited  0.000
 8.7558 Customer01: Here I am
10.6770 Customer02: Here I am
22.7622 Customer03: Here I am
```

```
32.7477 Customer04: Here I am
55.0607 Customer00: Finished
55.0607 Customer01: Waited 46.305
61.8905 Customer01: Finished
61.8905 Customer02: Waited 51.213
83.7556 Customer02: Finished
83.7556 Customer03: Waited 60.993
108.7794 Customer03: Finished
108.7794 Customer04: Waited 76.032
118.8254 Customer04: Finished
```

This model with random arrivals and exponential service times is known as the M/M/1 queue and can be solved analytically, for example to calculate the theoretical steady-state mean waiting time.

Several Counters

When we introduce several counters we must decide the queue discipline. Are customers going to make one queue or are they going to form separate queues in front of each counter? Then there are complications - will they be allowed to switch lanes (jockey)? We look first at a single-queue with several counters then at several isolated queues.

Several Counters but a Single Queue

The bank model with customers who arrive randomly to be served at a group of counters taking a random time for service. A single queue is assumed.

```
 1 #!/usr/bin/env python
 2 """ bank09: Several Counters but a Single Queue"""
 3 from SimPy.Simulation  import *
 4 from random import expovariate, seed
 5
 6 class Source(Process):
 7     """ Source generates customers randomly"""
 8
 9     def generate(self,number,interval):
10         for i in range(number):
11             c = Customer(name = "Customer%02d"%(i,))
12             activate(c,c.visit(timeInBank=12.0))
13             t = expovariate(1.0/interval)
14             yield hold,self,t
15
16 class Customer(Process):
17     """ Customer arrives, is served and leaves """
18
19     def visit(self,timeInBank=0):
20         arrive=now()
21         print "%7.4f %s: Here I am "%(now(),self.name)
22         yield request,self,counter
23         wait=now()-arrive
24         print "%7.4f %s: Waited %6.3f"%(now(),self.name,wait)
25         tib = expovariate(1.0/timeInBank)
26         yield hold,self,tib
```

```
27         yield release,self,counter
28         print "%7.4f %s: Finished"%(now(),self.name)
29
30 def model(theseed):
31     global counter
32     seed(theseed)
33     counter = Resource(name="Clerk",capacity = 2)
34
35     initialize()
36     source = Source('Source')
37     activate(source,source.generate(5,interval=10.0),0.0)
38     simulate(until=400.0)
39
40 model(theseed=12345)
```

The *only* difference between this model and the single-server model is in line 33. I have increased the capacity of the *counter* resource to 2 and, because both clerks cannot be called *Karen*, I have specified the name *Clerk*. The waiting times, however, are different. The second server has cut the waiting times:

```
 0.0000 Customer00: Here I am
 0.0000 Customer00: Waited  0.000
 8.7558 Customer01: Here I am
 8.7558 Customer01: Waited  0.000
10.6770 Customer02: Here I am
20.6626 Customer03: Here I am
23.2580 Customer01: Finished
23.2580 Customer02: Waited 12.581
30.0878 Customer02: Finished
30.0878 Customer03: Waited  9.425
37.0790 Customer04: Here I am
51.9528 Customer03: Finished
51.9528 Customer04: Waited 14.874
55.0607 Customer00: Finished
61.9988 Customer04: Finished
```

Several Counters with individual queues

Each counter is now assumed to have its own queue. The obvious modelling technique therefore is to make each counter a separate resource. Then we have to decide how a customer will choose which queue to join. In practice, a customer will join the counter with the fewest customers. An alternative (and one that can be handled analytically) is to allow him to join a queue "at random". This may mean that one queue is long while another server is idle. Here we allow the customer to choose the counter with the fewest customers. If there are more than one that satisfies this criterion he will choose the first.

```
1 #!/usr/bin/env python
2 """ bank10: Several Counters with individual queues"""
3 from SimPy.Simulation  import *
4 from random import expovariate,seed
5
6 class Source(Process):
7     """ Source generates customers randomly"""
```

```
 8
 9     def generate(self,number,interval):
10         for i in range(number):
11             c = Customer(name = "Customer%02d"%(i,))
12             activate(c,c.visit(timeInBank=12.0))
13             t = expovariate(1.0/interval)
14             yield hold,self,t
15
16 def NoInSystem(R):
17     """ The number of customers in the resource R
18     in waitQ and active Q"""
19     return (len(R.waitQ)+len(R.activeQ))
20
21 class Customer(Process):
22     """ Customer arrives, is served and leaves """
23
24     def visit(self,timeInBank=0):
25         arrive=now()
26         Qlength = [NoInSystem(counter[i]) for i in range(Nc)]
27         print "%7.4f %s: Here I am. %s   "%(now(),self.name,Qlength)
28         for i in range(Nc):
29             if Qlength[i] ==0 or Qlength[i]==min(Qlength):
30                 join =i ; break
31
32         yield request,self,counter[join]
33         wait=now()-arrive
34         print "%7.4f %s: Waited %6.3f"%(now(),self.name,wait)
35         tib = expovariate(1.0/timeInBank)
36         yield hold,self,tib
37         yield release,self,counter[join]
38
39         print "%7.4f %s: Finished    "%(now(),self.name)
40
41 def model(theseed):
42     global Nc,counter
43     seed(theseed)
44     Nc = 2
45     counter = [Resource(name="Clerk0"),Resource(name="Clerk1")]
46
47     initialize()
48     source = Source('Source')
49     activate(source,source.generate(number=5,interval=10.0),0.0)
50     simulate(until=400.0)
51
52 model(theseed = 3939393)
```

We need to find the total number of customers at each counter. To make the programming clearer, I define a Python function, *NoInSystem(R)* (lines 16-19) which returns the sum of the number waiting and the number being served for a particular counter, *R*. This function is used in line 26 to get a list of the numbers at each counter. I have modified the trace printout, line 27 to display the state of the system when the customer arrives. It is then obvious which counter he will join. We then choose the shortest queue in lines 28-30 (the variable *join*).

The remaining program is the same as before.

```
 0.0000 Customer00: Here I am. [0, 0]
 0.0000 Customer00: Waited  0.000
 7.7777 Customer01: Here I am. [1, 0]
 7.7777 Customer01: Waited  0.000
13.5590 Customer00: Finished
18.8841 Customer02: Here I am. [0, 1]
18.8841 Customer02: Waited  0.000
22.2894 Customer03: Here I am. [1, 1]
22.9091 Customer02: Finished
22.9091 Customer03: Waited  0.620
24.1283 Customer04: Here I am. [1, 1]
28.4592 Customer01: Finished
40.5521 Customer03: Finished
40.5521 Customer04: Waited 16.424
42.5870 Customer04: Finished
```

The results show how the customers choose the counter with the smallest number. For this sample, the fifth customer waits 16.424 minutes which is longer than he waited in the previous 2-server model, There are, however, too few arrivals in these runs, limited to five customers, to draw any general conclusions about the relative efficiencies of the two systems.

Monitors and Gathering Statistics

The traces of output that have been displayed so far are valuable for checking that the simulation is operating correctly but would become too much if we simulate a whole day. We do need to get results from our simulation to answer the original questions. What, then, is the best way to gather results?

One way is to analyze the traces elsewhere, piping the trace output, or a modified version of it, into a *real* statistical program such as *R* for statistical analysis, or into a file for later examination by a spreadsheet.

Another useful way of dealing with the results is to provide a graphical output. I do not have space to examine this thoroughly here (but see the comments in Final Remarks). *SimPy* offers an easy way to gather a few simple statistics such as averages: the *Monitor* class. These record the values of chosen variables as time series.

The bank with a Monitor

To demonstrate the use of a *Monitor* let us observe the average waiting times for our customers. In the following program I have removed the old trace statements because I will run the simulations for many more arrivals. In practice, I would make the printouts controlled by a variable, say, *TRACE* which is set in *model()*. This would aid in debugging and would not complicate the data analysis. However, I have not done this here.

The bank model has customers who arrive randomly to be served at two counters taking a random time for service. A customer chooses the shortest queue to join. A Monitor variable records the waiting time of customers.

```
1 #!/usr/bin/env python
2 """ bank11: The bank with a Monitor"""
3 from SimPy.Simulation  import *
4 from random import expovariate,seed
5
```

```
 6 class Source(Process):
 7     """ Source generates customers randomly"""
 8
 9     def generate(self,number,interval):
10         for i in range(number):
11             c = Customer(name = "Customer%02d"%(i,))
12             activate(c,c.visit(timeInBank=12.0))
13             t = expovariate(1.0/interval)
14             yield hold,self,t
15
16 def NoInSystem(R):
17     """ The number of customers in the resource R
18     in waitQ and active Q"""
19     return (len(R.waitQ)+len(R.activeQ))
20
21 class Customer(Process):
22     """ Customer arrives, is served and leaves """
23
24     def visit(self,timeInBank=0):
25         arrive=now()
26         Qlength = [NoInSystem(counter[i]) for i in range(Nc)]
27         for i in range(Nc):
28             if Qlength[i] ==0 or Qlength[i]==min(Qlength): join =i ; break
29         yield request,self,counter[join]
30         wait=now()-arrive
31         waitMonitor.observe(wait)
32         tib = expovariate(1.0/timeInBank)
33         yield hold,self,tib
34         yield release,self,counter[join]
35
36 def model(theseed=393939):
37     global Nc,counter,waitMonitor
38     seed(theseed)
39     Nc = 2
40     counter = [Resource(name="Clerk0"),Resource(name="Clerk1")]
41
42     waitMonitor = Monitor()
43
44     initialize()
45     source = Source('Source')
46     activate(source,source.generate(number=50,interval=10.0),0.0)
47     simulate(until=2000.0)
48
49     return (waitMonitor.count(),waitMonitor.mean())
50
51 result = model(393939)
52 print "Average wait for %4d completions was %6.2f"% result
```

The monitor *waitMonitor* is created in line 42 and declared to be global in line 37. It *observes* the waiting times in line 31. At the end of the *model* function, line 49, the number observed and the mean waiting time are returned.

model is run in line 51 and the results printed in line 52. Here I ran 50 customers (in the call of

generate in line 46) and have increased the *until* argument in line 47 to 2000.

```
Average wait for   50 completions was   3.15
```

The average waiting time for 50 customers in a 2-counter, 2-queue system is more reliable than the times we measured before but it is not very convincing. We should replicate the runs using different random number seeds.

Multiple runs of the bank with a Monitor

The advantage of the way the programs have been set up with the main part of the program in a *model()* routine will now become apparent. We will not need to make any changes to run a series of replications:

Just as before customers who arrive randomly to be served at two counters taking a random time for service. A customer chooses the shortest queue to join. A Monitor variable observes the waiting times. A number of replications are. The following partial listing only shows the *model* and its call.

```
50
51 # now carry out a number of replications
52 theseed = [393939,31555999,777999555,319999771]
53 for i in range(4):
54     result = model(theseed[i])
55     print "Average wait for %4d completions was %6.2f"% result
56
```

The change is in lines 52-55. *model()* is now run for four different random-number seeds to get a set of replications. The results are printed with a limited number of significant digits.

```
Average wait for   50 completions was   3.15
Average wait for   50 completions was   6.47
Average wait for   50 completions was   3.58
Average wait for   50 completions was  11.24
```

The results show variation. Remember, though, that the system is only operating for 50 customers so the system may not be in steady-state.

Final Remarks

This introduction is too long and the examples are getting longer. There is much more to say about simulation with *SimPy* but no space. I finish with a list of topics for other documents:

- **GUI input**. Graphical input of simulation parameters could be an advantage in many cases. *SimPy* allows this and programs using these facilities have been developed (see, for example, program *MM1.py* in the examples in the *SimPy* distribution)
- **Graphical Output**. Similarly, graphical output of results can also be of value, not least in debugging simulation programs and checking for steady-state conditions.
- **Statistical Output**. The *Monitor* class is of some value in presenting results but more powerful methods of analysis are often needed. One solution is to output a trace and read that into a large-scale statistical system such as *R*.
- **Advanced synchronization/scheduling commands**. *SimPy* 1.5 introduces synchronization by events and signals.

- **Interactions between processes**. Often the simple-minded *Resource* model is insufficient. For example a tug may be needed to carry out many tasks in a port model but demands on it may have to be queued. Here we can drop back to the older, *Simula* inspired, methods of process interaction. In these methods a process suspends itself after joining a waiting queue (modelled as a simple *Python* list) and is reactivated when the tug, say, is available.

Acknowledgments

I thank those developers and users of SimPy who have improved this document by sending their comments. I will be grateful for further corrections or suggestions. Could you send them to me: *vignaux* at *users.sourceforge.net*.

References

- Python website: http://www.Python.org
- SimPy website: http://sourceforge.net/projects/simpy