**Boston University** **Fall 2018**

# ECE Department

**Computer Communication Networks (EC 541)**

**An simple SimPy introduction by examples**

1. **Introduction: purpose of the document**

SimPy is “a process-based discrete-event simulation framework based on standard Python”. For this course, SimPy is particularly useful for building a queue to monitor the dynamic and statistical parameters, and also good to simulate different strategies in Bitcoin miners’ behavior. This document is meant to give a brief and practical introduction to the basic components of SimPy like ‘process’, ‘events’, ‘resource’… An assumption is made that the students already have a basic concepts of Python programming like generator and class.

We are starting from a simple example of a process reporting time at random intervals and two concurrent processes reporting time at random intervals. Then we give a more realistic turn-based game with two players, introducing the concept ‘shared event’ and ‘interruption’. Lastly, we build the standard M/M/1 queue, introducing the ‘resource’ used here as a server. The SimPy version we are using is 3.010 and the python version is 3.6.3.

1. **Basic random process: time reporting**

This example shows a traditional way of building a process when the behaviors of the program are determined at random time intervals. A ‘for’ loop can be used to build the sequence of events and the time intervals can be generated either in each step or generated at the beginning at the program. SimPy is not necessary in this basic scenario.

|  |
| --- |
| import simpy as sp  import time  import numpy as np  N = 5  step = 0  def syn\_proc():  for step in range(N):  print('0.', step, ' seconds elapsed.')  time.sleep(np.random.uniform(0,1))  syn\_proc() |

When we need to simulate a more complex situation like two parallel random processes, SimPy may have its position here. First we initialize a ‘enviroment’. The processes like ‘env.process(proc1(env)) ’ are decleared but not ‘run in real time enviroment’ until ‘env.run(until = N)’. The keyword 'yield' is a python keyword for generator and a 'process' in SimPy must be a python generator in SimPy lib. Details about 'yield' can be found here: <https://wiki.python.org/moin/Generators>. The codes after ‘yield’ ‘runs’ until the process finishes in the ‘environment’. This could be clearer in the turn-based game example.

|  |
| --- |
| env = sp.Environment()  def proc1(env):  while True:  yield env.timeout(np.random.uniform(0,1))  print('proc1: ', env.now)  def proc2(env):  while True:  yield env.timeout(np.random.uniform(3,4))  print('proc2: ',env.now)  env.process(proc1(env))  env.process(proc2(env))  env.run(until = N) |

1. **Process sharing an event: turn-based game:**

Normally parallel processes have more complex behaviors than reporting the time. The turn-based game has two players. A class ‘game\_turnBased’ is defined. The ‘main\_proc’ is declared as a method to run the rounds in the for loop and a ‘finishEvent’ is restarted in each round. In the player’s process ‘proc1’ and ‘proc2’, after each player ‘finishes’ the round, he checks if the ‘finishEvent’ is triggered or not. If it is not triggered, he wins the round and trigger the event by ‘self.finishEvent.succeed()’. Each round finishes after both players finish the round------‘yield proc1 & proc2’.

|  |
| --- |
| class game\_turnBased:  def \_\_init\_\_(self):  print('\n\npart 3')  self.env = sp.Environment()  self.finishEvent = self.env.event()  self.action = self.env.process(self.main\_proc()) # process decleared, wait for env.run() to start  self.N\_rounds = 10    def main\_proc(self):  print('starts')  for i in range(self.N\_rounds):  print('Round', i, '\*\*\*\*\*')  yield self.env.process(self.proc1(self.env, i)) & self.env.process(self.proc2(self.env, i))# one round should finish until the two processes both return results, win or failed  self.finishEvent = self.env.event()# restart the round  def proc1(self, env, Rd):  yield env.timeout(np.random.uniform(0,1))  if not self.finishEvent.triggered: #to see if the others have already decleared finish  self.finishEvent.succeed()  print('player1 won round ', Rd,' at: ', env.now)  else:  print('player1 failed round ', Rd,' at: ', env.now)  def proc2(self, env, Rd):  yield env.timeout(np.random.uniform(0,1))  if not self.finishEvent.triggered: #to see if the others have already decleared finish  self.finishEvent.succeed()  print('player2 won round ', Rd,' at: ', env.now)  else:  print('player2 failed round ', Rd,' at: ', env.now)  onegame = game\_turnBased()# the game should be processed in the environment  onegame.env.run() |

As in the last part, this situation can be simulated as easy as generating two lists of random numbers to see the statistical results without bothering using SimPy. But things change when each round ends when either player finishes, where we introduce the ‘interrupt’. When either player wins the round, it interrupts the other process by ‘self.p2.interrupt()’. This structure can be implemented when we simulate the selfish strategy for Bitcoin mining. The selfish player can interrupt the other miners or not when they mined a block.

|  |
| --- |
| class game\_turnBased\_inter:  def \_\_init\_\_(self):  print('\n\npart 4')  self.env = sp.Environment()  self.action = self.env.process(self.main\_proc()) # process decleared, wait for env.run()  self.p1 = None #player1's process for each round  self.p2 = None #player1's process for each round  self.N\_rounds = 10    def main\_proc(self):  print('starts')  for i in range(self.N\_rounds):  print('Round', i, '\*\*\*\*\*')  self.p1 = self.env.process(self.proc1(self.env, i))  self.p2 = self.env.process(self.proc2(self.env, i))  yield self.p1 & self.p2 # one round should finish until the two processes both return results, win or failed, now at the same time  def proc1(self, env, Rd):  try:  yield env.timeout(np.random.uniform(0,1))  print('player1 won round ', Rd,' at: ', env.now)  self.p2.interrupt() #now interrupt the other process  except sp.Interrupt:  print('player1 failed round ', Rd,' at: ', env.now)  def proc2(self, env, Rd):  try:  yield env.timeout(np.random.uniform(0,1))  print('player2 won round ', Rd,' at: ', env.now)  self.p1.interrupt() #now interrupt the other process  except sp.Interrupt:  print('player2 failed round ', Rd,' at: ', env.now)    onegame2 = game\_turnBased\_inter()# the game should be processed in the environment  onegame2.env.run() |

1. **Queueing system**

A built-in concept of ‘resource’ is naturally suitable to simulate the queue. Fist we define a class of arrivals, which has attributes of ‘ID’, ‘arrivalTime’, ‘serviceTime’, ‘action’. A customer takes the action of ‘joinQueue’, which is a SimPy process. In the process, a customer requests the resource by ‘with self.res.request() as req: yield req’. When it gets the resource, the ‘waitTime’ for each customer is determined.

The class ‘queueSim’ is also defined here for generality. The resource is initialized here. In the ‘action’, a ‘for’ loop is used to generate ‘N’ customers and records each customer’s waiting time.

|  |
| --- |
| Lambda = 0.5  Mu = 1  N = 10000  class arrivals:  def \_\_init\_\_(self, ID, env, res):  self.ID = ID  self.arrivalTime = env.now  self.waitTime = 0  self.serviceTime = np.random.exponential(1/Mu)  self.action = env.process(self.joinQueue(env))  self.res = res  # print(ID, 'arrives at: ', self.arrivalTime)    def joinQueue(self, env):  with self.res.request() as req:  yield req #wait until getting into the server  self.waitTime = env.now - self.arrivalTime  # print(self.ID, 'got server at: ', env.now)  # print(self.ID, 'waited for: ', self.waitTime)  yield env.timeout(self.serviceTime) #the service time is exponentially distributed  self.res.release(req) # release the server  class queueSim:  def \_\_init\_\_(self):  self.env = sp.Environment()  self.customers = []  self.action = self.env.process(self.action())  self.averWaitTime = 0  self.res = sp.Resource(self.env, capacity=1) #the server has one slot  print('\n\npart 5')  def action(self):  for i in range(N):  yield self.env.timeout(np.random.exponential(1/Lambda)) #the time interval of arrivals is exponentially distributed  self.customers.append(arrivals(i, self.env, self.res))  for c in self.customers:  self.averWaitTime += c.waitTime    q1 = queueSim()  q1.env.run()  print('Average waiting time is: ', q1.averWaitTime/N) |

1. **Reference**
2. <https://wiki.python.org/moin/Generators>
3. <https://pythonhosted.org/SimPy/Tutorials/TheBank.html>