Discrete Event Simulation. It's Easy with SimPy!

Article · February 2018

CITATIONS
3

READS
9,991

1 author:

Dmitry Zinoviev
Suffolk University
111 PUBLICATIONS **878** CITATIONS

SEE PROFILE

The Pragmatic Bookshelf

PragPub

Keeping It Light

IN THIS ISSUE:

* Dmitry Zinoviev on how easy discrete event simulation can be

Issue #104
February 2018

Contents

FEATURES

COLUMNS

DEPARTMENTS

Published monthly in PDF, mobi, and epub formats by The Pragmatic Programmers, LLC, Dallas, TX, and Raleigh, NC. E-Mail webmaster@swaine.com. The editor is Michael Swaine (michael@pragprog.com). Visit us at http://pragprog.com for the lowdown on our books, screencasts, training, forums, and more.

ISSN: 1948-3562

Discrete Event Simulation

It's Easy with SimPy!

by Dmitry Zinoviev

Computer modeling and simulation is the art of bringing to life systems and behaviors that otherwise are prohibitively expensive, unethical, or just impossible to build.

Computer modeling and simulation (M&S) is a priceless art of bringing to "life" systems and behaviors that otherwise are prohibitively expensive, unethical, or just impossible to build, such as a Mars rover, a centaur, or a cruel autocratic regime based on slave ownership.

An M&S project consists of several standard steps:

- First, an M&S specialist builds a *model*. The most common model types are:
  - Mathematical models that represent the relationships between the system variables as one or more linear or differential equations (Ohm's law is an example of a linear model), and
  - Discrete event models that represent the system as a collection of finite states and transitions between them caused by internal or external events. The simulation time "jumps" from one event to the next one, and there is a timeless void in between.
- Then, the model is *simulated* to obtain execution traces. The traces describe the evolution of some or all system variables over time.
- The traces are often *visualized* because a picture is worth 1,000 words.
- If the goal of the project is to discover the optimal values of the system parameters, then *optimization* is the next step. During optimization, the system model is repeatedly simulated with different parameters to minimize or maximize a predefined goal function. The optimization routine may use constrained or unconstrained functional optimization, curve fitting, simulated annealing, or brute force.

Once implemented, debugged, simulated, and optimized, a model can be used to evaluate, build, and even replace the original system.

There is a plethora of general purpose M&S software, both commercial and Open Source, such as OpenModelica [U1], Ptolemy [U2], Simulink [U3], and even Simula [U4] — a complete computer programming language designed to support simulation. In this narrowly focused article, I will show you how to do simple discrete event simulation with SimPy [U5] — an M&S module written in pure Python.

Installing SimPy

The easiest (and the only recommended) way to install SimPy is via pip:

```
pip install simpy
=>...
=>Successfully installed simpy-3.0.10
```

At the time of writing, simpy-3.0.10 is the most recent version of SimPy, and I will use it for all examples.

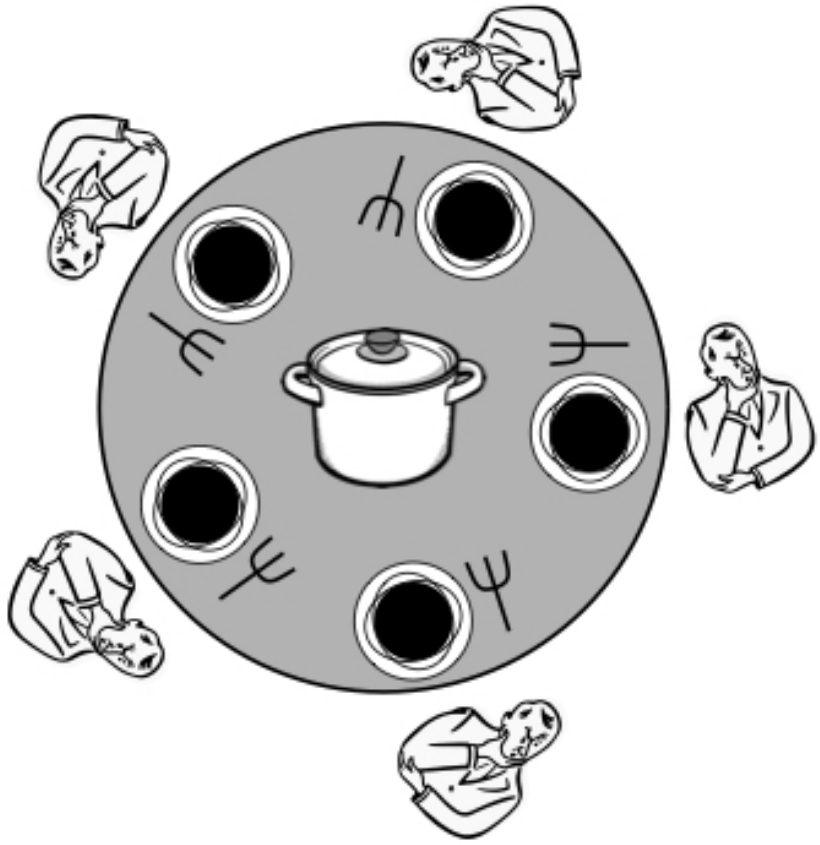
To check if SimPy was successfully installed, open a Python shell and import simpy.

Assuming you see no error messages, the next goal is to find a suitable system to model and simulate. Why not revisit the good old Dining Philosophers?

Dining Philosophers

In brief, five really fine Philosophers sit around a table with an enormous dish of freshly made spaghetti at the center. Most of the time they think, but now and then a Philosopher becomes hungry, at which point he collects two forks: a left one and a right one — and starts eating the pasta. Having refueled himself, he puts down the forks and returns to thinking.

Just like all fine philosophers, the five dining philosophers barely make ends meet and can afford only five forks, which prevents more than two of them from eating simultaneously. In addition to being poor, the philosophers are also stubborn and uncooperative: once they pick up one of the forks, they never put it down until they have a chance to eat. Think what happens if all the five guys pick up their left forks at about the same time. (A hint: they starve.)

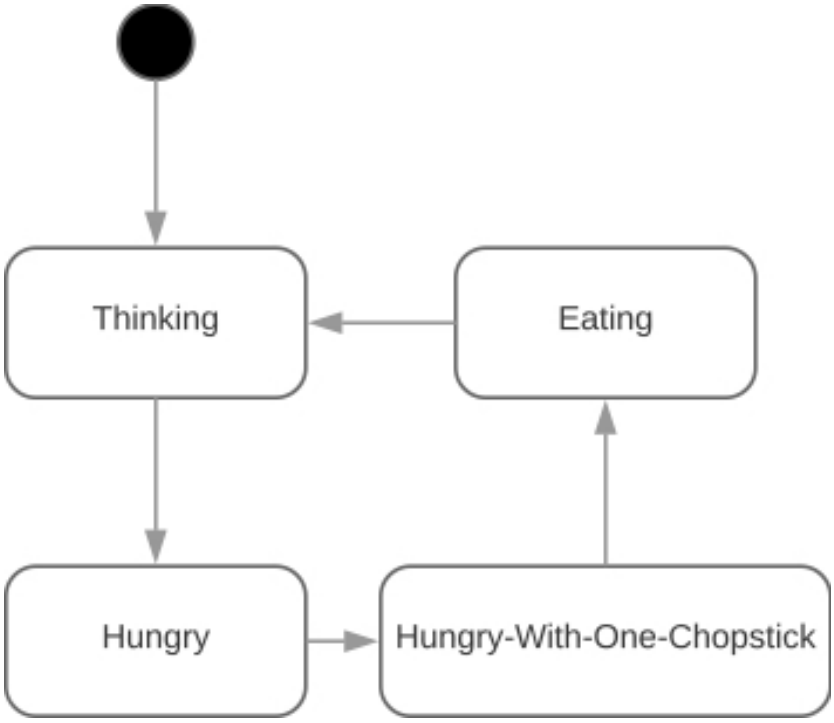


The five dining philosophers

Disclaimer: Not being of Italian heritage, I never understood the need for two forks to eat pasta. Without the loss of generality, for the rest of the story, I will refer to forks as chopsticks and a dish of pasta as bowl of rice.

Edsger Dijkstra introduced the dining philosophers problem in 1965, and ever since it has been a gold standard test framework for concurrent algorithm designers. The problem is simple and well-understood, which makes it an excellent candidate for an introductory simulation project. I will model each philosopher and the interactions between the philosophers and the chopsticks.

From the discrete event simulation point of view, a philosopher is a finite state machine with four states: Thinking (the initial state), Hungry, Hungry-With-One-Chopstick, and Eating. There is no final state in the diagram, as the philosophers are never supposed to break the cycle.



State transition diagram of a dining philosopher

The transitions between the states are caused by external or internal events. A thinking philosopher becomes hungry *after some time* — on timeout. The time is based on some internal philosopher's considerations unknown to us. I can model it as an exponentially distributed random variable with the mean value T0. The transition from the Eating state to the Thinking state is triggered internally, too. I can model it as another exponentially distributed random variable with the mean value T1.

(Note: It is a common and dangerous mistake to treat times as normally distributed random variables — in the first place, due to the unbounded nature of a normal distribution.)

The other two transitions in the state transition diagram result from external events: the availability of the chopsticks. A philosopher becomes Hungry-With-One-Chopstick when he manages to collect the first chopstick, and he becomes Eating when the second chopstick becomes available.

Last but not least, I assume that a philosopher spends some small but finite time DT in the Hungry-With-One-Chopstick state even if the other chopstick is readily available. If he picks up both chopsticks at once, the system never ends

up in a deadlock, which is practically desirable — but not very exciting from the S&M point of view.

Simulating the "Classical" Dining Philosophers

Enter SimPy.

At the core of any SimPy model, there is an environment (simpy.Environment). The environment, among other things, provides functions for the continuous or step-by-step simulation of the models, dispatches events, and keeps track of the current simulation time. I start by importing the necessary modules and creating a new simulation environment:

```
import simpy
import random
env = simpy.Environment()
```

I will pass the environment to all other entities involved in the model — for example, to the chopsticks.

Using the S&M terminology, a chopstick is a *renewable resource*. It is managed by the environment and must be created before the first use. In the course of the simulation, a resource can be requested, allocated, and released. SimPy provides three types of resources: "vanilla" resources (simpy.Resource, simpy.PriorityResource, simpy.PreemptiveResource), containers (simpy.Container), and stores (simpy.Store, simpy.PriorityStore, simpy.FilterStore). I will use the basic simpy.Resource to model a chopstick, and look at the other resources later.

Each resource has a capacity that defines how many resource users can share it. If the number of requests exceeds the capacity, SimPy blocks the outstanding requests until some of the users release the resource. For the sanitary and convenience reasons, a chopstick should not be shared; I set its capacity to 1 (which makes it a *mutually-exclusive* resource) and create five chopsticks:

```
N = 5
chopsticks = [simpy.Resource(env, capacity=1) for i in range(N)]
```

My next step is to define the philosophers. Unlike the chopsticks, the philosophers are active. In SimPy, active entities are known as processes. (Not to be confused with the operating system processes!) A process is a Python generator that yields discrete events. (If your knowledge of the yield keyword is a little rusty, you may want to refresh it before reading further.) The simulation environment has a method env.process that registers the processes. In the following code, I declare the process function run_the_party as a Python class method. I create an event generator and register it after initializing the class and instance variables.

```
class Philosopher():
    T0 = 10 # Mean thinking time
    T1 = 10 # Mean eating time
    DT = 1 # Time to pick the other chopstick
    def __init__(self, env, chopsticks, my_id, DIAG = False):
        self.env = env
        self.chopsticks = chopsticks
        self.id = my_id
        self.waiting = 0
        self.DIAG = DIAG
        # Register the process with the environment
        env.process(self.run_the_party())
    def get_hungry(self): # Request the resources
        yield # Do nothing so far
    def run_the_party(self): # Do everything...
        yield # ...but do nothing so far
    def diag(self, message): # Diagnostic routine
        if self.DIAG:
            print("P{} {} @{}".format(self.id, message, self.env.now))
```

Note that the class constructor takes several parameters: the simulation environment, the chopstick resources, and the philosopher's id (for diagnostics). Also note that the current simulation time is available as an attribute of the environment, env.now.

My philosophers do not yield any events and are dysfunctional, but I already can instantiate them and assign two chopsticks to each philosopher:

```
philosophers = [Philosopher(env,
                            (chopsticks[i], chopsticks[(i + 1) % N]), i)
                for i in range(N)]
```

The event generator run_the_party yields events of three types: timeout, request, and release. A call to env.timeout(tau) blocks the generator for tau time units. I use delays to model the thinking and eating activities. A call to resource.request blocks the generator until the resource becomes available, marks the resource as allocated to the caller, and returns the request id. Finally, a call to resource.release(rq) deallocates the resource associated with the request id rq. It does not block the generator.

The following code is a translation of the Philosopher State to the language of SimPy. It consists of two methods: implementing the main process and a subprocess.

```
class Philosopher():
    ...
    def get_hungry(self):
        start_waiting = self.env.now
        self.diag("requested chopstick")
        rq1 = self.chopsticks[0].request()
        yield rq1
        self.diag("obtained chopstick")
        yield self.env.timeout(self.DT)
        self.diag("requested another chopstick")
        rq2 = self.chopsticks[1].request()
        yield rq2
        self.diag("obtained another chopstick")
        self.waiting += self.env.now - start_waiting
        return rq1, rq2
    def run_the_party(self):
        while True:
            # Thinking
            thinking_delay = random.expovariate(1 / self.T0)
            yield self.env.timeout(thinking_delay)
            # Getting hungry
            get_hungry_p = self.env.process(self.get_hungry())
            rq1, rq2 = yield get_hungry_p
            # Eating
            eating_delay = random.expovariate(1 / self.T1)
            yield self.env.timeout(eating_delay)
            # Done eating, put down the chopsticks
            self.chopsticks[0].release(rq1)
            self.chopsticks[1].release(rq2)
            self.diag("released the chopsticks")
```

The method get_hungry is a subprocess — a process within a process. It hides the technicalities of getting hungry (get one chopstick, get another chopstick, etc.), letting me concentrate on the conceptual flow of the party. The main process is suspended until the subprocess yields all events. Note that the subprocess method returns the chopstick request handlers — I use them later to release the resources.

My model is ready to run. I can simulate it step by step (or, rather, event by event) by calling env.step; or for the first t time units by calling env.run(until=t); or until the simulation naturally stops by calling env.run without any parameters. Here's a sample output of the script:

```
env.run()
=>P1 requested chopstick @0.10843721582414197
=>P1 obtained chopstick @0.10843721582414197
=>P1 requested another chopstick @1.108437215824142
=>P1 obtained another chopstick @1.108437215824142
=>P0 requested chopstick @6.642119333001387
=>P0 obtained chopstick @6.642119333001387
=>P0 requested another chopstick @7.642119333001387
=>P2 requested chopstick @7.898070765597094
=>...
=>P2 obtained chopstick @45632.71748485687
=>P1 requested another chopstick @45632.721825172666
=>P4 requested another chopstick @45632.94176319329
=>P3 requested another chopstick @45633.53086921637
=>P2 requested another chopstick @45633.71748485687
```

At this point, the simulation suddenly stops: the system is in a deadlock state where each philosopher holds one chopstick and waits for the other one. There

is no standard way to detect deadlocks in SimPy. An indirect solution is to look at the counts of requests or actual requests granted for each resource (they are stored in the attributes resource.count and resource.users, respectively). If at least two counts are greater than zero and the simulation stopped, the system is likely to be deadlocked:

```
[f.count for f in chopsticks]
=>[1, 1, 1, 1, 1]
```

Incidentally, the requests that have not been granted yet can be found in the attribute resource.queue.

One of the well-known solutions to a deadlock is to allocate resources in some globally defined order (say, in the order of their numerical ids or memory references). The philosophers will never starve if I add Python id-based sorting to the second line of the class initializer:

```
self.chopsticks = sorted(chopsticks, key=id)
```

Gathering and Visualizing Statistics

Usually the goal of an M&S project is to verify the correctness of the modeled system and get some sense of its performance. The most common performance characteristics include waiting time, turnaround time, response time, and resource utilization. As an exercise, let's estimate waiting time — the amount of time spent by a process waiting for the requested resources (ready to run but not running).

Unlike other M&S software, SimPy does not provide tools for gathering and visualizing simulation statistics — and, to be honest, it does not have to. Being a part of the Python ecosystem, SimPy can be easily integrated with statistics, NumPy, Pandas, SciPy, matplotlib, and other plotting, statistical, and general number-crunching packages.

I measure the mean waiting time of the philosophers by instrumenting the Philosopher class with a time-counting variable waiting.

To compare the waiting times or any other performance metrics for different systems (say, the dining parties with various numbers of philosophers), I need a function that prepares a model, makes a simulation run, and returns the appropriate performance values:

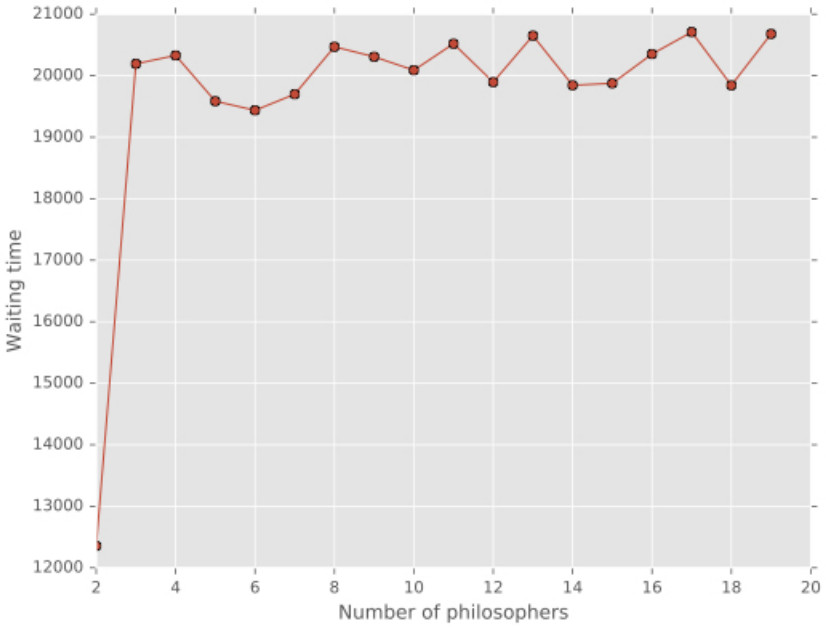
```
def simulate(n, t):
    """
    Simulate the system of n philosophers for up to t time units.
    Return the average waiting time.
    """
    env = simpy.Environment()
    chopsticks = [simpy.Resource(env, capacity=1) for i in range(n)]
    philosophers = [
        Philosopher(env, chopsticks[i], chopsticks[(i + 1) % n], i)
        for i in range(n)]
    env.run(until=t)
    return sum(ph.waiting for ph in philosophers) / n
```

I will call this all-in-one function for different values of n and plot the results with matplotlib — the ubiquitous Python plotting engine. Note that my

preferred plotting style, ggplot, is a reference to the rival programming language R.

```
# Set up the plotting system
import matplotlib.pyplot as plt
import matplotlib
matplotlib.style.use("ggplot")
# Simulate
N = 20
X = range(2, N)
Y = [simulate(n, 50000) for n in X]
# Plot
plt.plot(X, Y, "-o")
plt.ylabel("Waiting time")
plt.xlabel("Number of philosophers")
plt.show()
```

The following figure shows the simulation results. As expected, the waiting time for two philosophers is shorter than for three philosophers. A less obvious result is that the waiting time for three or more philosophers is almost constant with respect to the party size.



Waiting time versus the number of philosophers

Any interpretation of these facts is beyond the scope of this story.

Adding a Container

No matter how you feel about the stubborn philosophers, the bottomless bowl of rice at the center of the table is unrealistic. Real rice ends soon, because it is a *consumable resource*. I can make the model somewhat less fictitious by modeling the rice bowl as a container with finite capacity. The amount of rice in the bowl will decrease by a fixed amount every time a philosopher eats from it. (For convenience, let's assume that the rice is atomically transferred to the philosopher's plate at the moment he collects both chopsticks.) I will call the new breed of philosophers "pseudo-philosophers," because genuine philosophers surely don't eat rice or anything material, but merely pretend to be eating.

A simpy.Container can yield two events of interest: put(amount) and get(amount). Both of them can block the generator if the container is nearly full or nearly empty, respectively. The capacity and current level of a container are stored in the namesake attributes, container.capacity and container.level.

Since I already have a Philosopher, the cheapest way to implement a pseudo-philosopher is to add an optional bowl parameter to the constructor and a request for a portion of rice to the get_hunry method if the bowl has been provided:

```
class Philosopher():
    T0 = 10 # Mean thinking time
    T1 = 10 # Mean eating time
    DT = 1 # Time to pick the other chopstick
    PORTION = 20 # Single meal size
    def __init__(self, env, chopsticks, my_id, bowl = None, DIAG = False):
        self.env = env
        self.chopsticks = sorted(chopsticks, key=id)
        self.id = my_id
        self.waiting = 0
        self.bowl = bowl
        self.DIAG = DIAG
        # Register the process with the environment
        env.process(self.run_the_party())
    def get_hungry(self):
        start_waiting = self.env.now
        self.diag("requested chopstick")
        rq1 = self.chopsticks[0].request()
        yield rq1
        self.diag("obtained chopstick")
        yield self.env.timeout(self.DT)
        self.diag("requested another chopstick")
        rq2 = self.chopsticks[1].request()
        yield rq2
        self.diag("obtained another chopstick")
        if self.bowl is not None:
            yield self.bowl.get(self.PORTION)
            self.diag("reserved food")
        self.waiting += self.env.now - start_waiting
        return rq1, rq2
    def run_the_party(self):
        ...
```

Incidentally, the new class design allows me to feed different philosophers from different bowls and even mix "true philosophers" and "pseudo-philosophers."

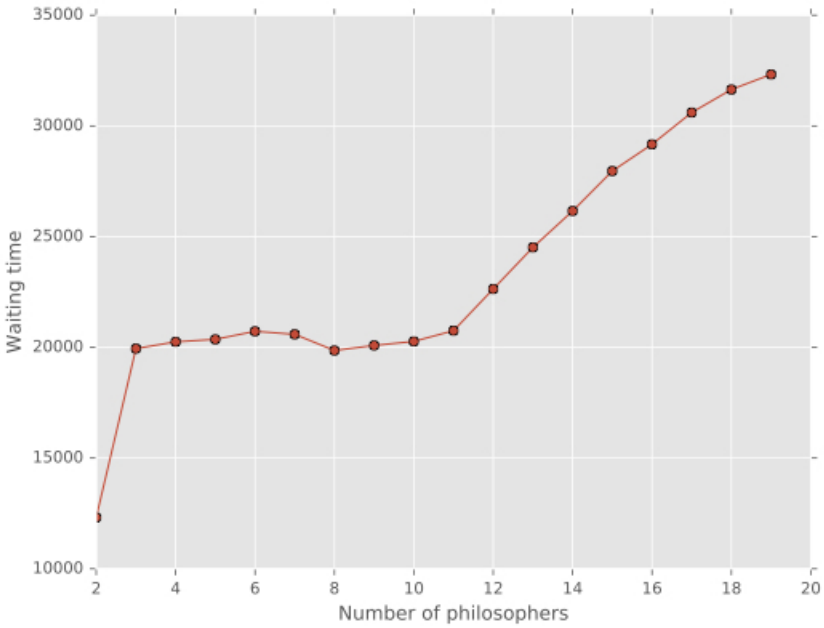
If the philosophers continue eating their rice, the bowl eventually becomes empty, and the party comes to an end. I will create another process — a chef — whose sole responsibility is to replenish the bowl every T2 time units:

```
class Chef():
    T2 = 150
    def __init__(self, env, bowl):
        self.env = env
        self.bowl = bowl
        env.process(self.replenish())
    def replenish(self):
       while True:
           yield self.env.timeout(self.T2)
           if self.bowl.level < self.bowl.capacity:
               yield self.bowl.put(self.bowl.capacity - self.bowl.level)
```

The updated function simulate has additional code for instantiating the bowl and the chef, and passes the bowl parameter to the philosophers:

```
def simulate(n, t):
    """
    Simulate the system of n philosophers for up to t time units.
    Return the average waiting time.
    """
    env = simpy.Environment()
    rice_bowl = simpy.Container(env, init=1000, capacity=1000)
    chef = Chef(env, rice_bowl)
    chopsticks = [simpy.Resource(env, capacity=1) for i in range(n)]
    philosophers = [
        Philosopher(env, (chopsticks[i], chopsticks[(i + 1) % n]), i,
                    rice_bowl)
        for i in range(n)]
    env.run(until=t)
    return sum(ph.waiting for ph in philosophers) / n
```

The next figure shows the waiting time for the new realistic model. As expected, there is almost no difference between the two models when the number of the dining philosophers is small (less than ten in my case): the chef is cooking fast enough to supply the food. When the number of the partygoers increases, the turnaround time of the chef becomes an issue and negatively affects the waiting time of the patrons.



Waiting time versus the number of pseudo-philosophers

Learning to Give Up

A hungry philosopher may not be that hungry, after all. If the central bowl has no rice in it and the chef is nowhere to be found, the philosopher may choose to call it a day: put down the chopsticks and go back to thinking. Presumably, he will have a chance to eat a supersized meal next time.

The problem with resource requests is that they are, in general, blocking. In fact, they can block the caller forever if the resource never becomes available (this is the reason for deadlocks). SimPy provides a mechanism for restricting the wait time by failing events and waiting for multiple events.

Introducing the new — impatient — kind of philosophers requires changing the methods get_hungry and run_the_party. First, I add constant MAX_WAIT that limits the waiting time to at most a half of the rice bowl replenishing time. Second, when a philosopher collects both chopsticks, the model fires two events: the container request (as before) and a timeout. The generator process is unblocked when *an*y of these events triggers, which is accomplished by combining the events with the logical or operator | (a vertical bar). The operator is a shortcut to the function simpy.events.AnyOf(env,events). If the modeling logic requires that *all* events be triggered before unblocking the process, use the logical and operator & or the function simpy.events.AllOf(env,events) instead.

(Note: If you are familiar with UML, note that | corresponds to the UML merge and & corresponds to the UML join.)

```
class Philosopher():
    ...
    MAX_WAIT = Chef.T2 / 2
    ...
    def get_hungry(self, meal_size):
        start_waiting = self.env.now
        self.diag("requested chopstick")
        rq1 = self.chopsticks[0].request()
        yield rq1
        self.diag("obtained chopstick")
        yield self.env.timeout(self.DT)
        self.diag("requested another chopstick")
        rq2 = self.chopsticks[1].request()
        yield rq2
        self.diag("obtained another chopstick")
        if self.bowl is not None:
            request = self.bowl.get(meal_size)
            yield request | self.env.timeout(self.MAX_WAIT)
            if request.processed:
                self.diag("reserved food")
            else: # Timeout
                self.diag("gave up")
                self.waiting += self.env.now - start_waiting
                yield simpy.Event(self.env).fail(ValueError(rq1, rq2))
        # Either no bowl or no timeout!
        self.waiting += self.env.now - start_waiting
        return rq1, rq2
```

After yielding the compound event, I check if SimPy processed the container request, by looking at the attribute request.processed. If it did, the food was

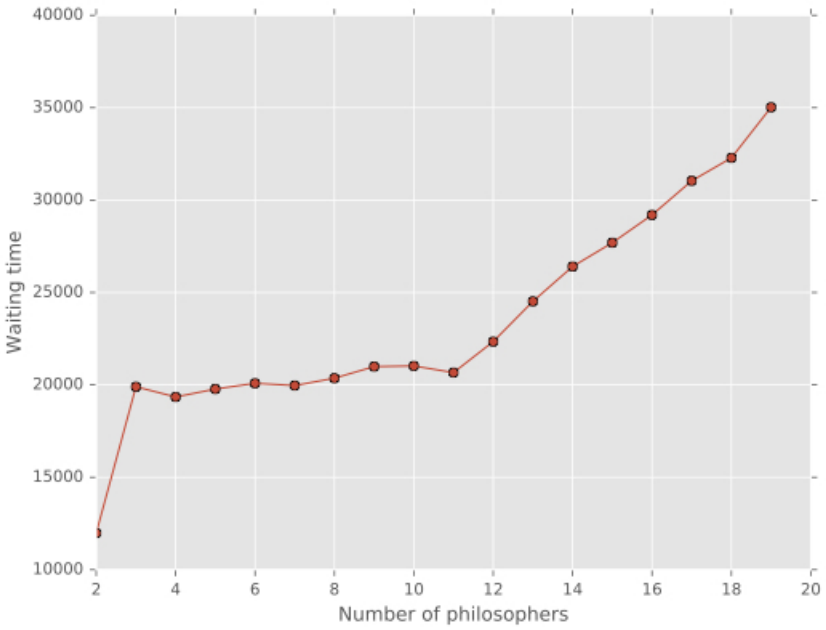
reserved, and the philosopher can start eating. Otherwise, the wait was interrupted by the timeout, and get_hungry must inform the parent process of the failure. SimPy allows a process to fail (and raise an exception) by yielding a namesake event. Incidentally, the raised ValueError exception takes the chopstick request handles as the parameters: when get_hungry fails, there is no other way to return the handles from it.

I instrumented the run_the_party method with simple meal size accounting. Every time a philosopher chooses not to wait for the chef, his next rice allowance is increased by one standard portion size. (Intuitively, his overall wait time increases, because some attempts are futile). I detect failed attempts and recover from the failures by catching the exception raised by get_hungry and extracting the chopstick request handles from the exception.

```
def run_the_party(self):
    meal_size = self.PORTION
    while True:
        yield self.env.timeout(random.expovariate(1 / self.T0))
        get_hungry_p = self.env.process(self.get_hungry(meal_size))
        try:
            rq1, rq2 = yield get_hungry_p
            yield self.env.timeout(random.expovariate(1 / self.T1))
            meal_size = self.PORTION
        except ValueError as values: # Timeout
            rq1, rq2 = values.args
            meal_size += self.PORTION
        self.chopsticks[0].release(rq1)
        self.chopsticks[1].release(rq2)
        self.diag("released chopsticks")
```

I can disable the new timeout feature by setting MAX_WAIT to a substantially large number — say, larger than T2.

The next figure shows the waiting time for the impatient philosophers. The difference between this model and the previous one is visible only when the number of partygoers is large enough.



Waiting time vs the number of impatient philosophers

Learning to Communicate

I am going to leave my good philosophers in the custody of the chef and use another classical example — a customer service counter — to explain how to model processes that interact directly rather than by sharing resources. In the new scenario, there is a stream of customers arriving with exponentially distributed delays and queueing into a line. (In the example below, the first ten customers are conveniently "generated" by a customer_generator_p process.) The clerk at the customer service counter takes the next customer from the line and services them in exactly SERVICE_DELAY time units. A customer is served successfully with probability 0.9. If there are no customers in the line, the clerk becomes idle ("falls asleep," if you like), but is reactivated ("woken up") by the next arriving customer.

SimPy provides two direct interaction mechanisms: process interruption and event-based synchronization. To interact, SimPy processes must have references to either each (simpy.events.Process, as returned by calls to env.process) or references to the events involved in the interaction (simpy.events.Event).

The following code illustrates both communication mechanisms. It begins with the imports and definitions of global variables. I use the Python standard double-ended queue collections.deque to model the service line.

SimPy uses Python exceptions to manage process interruptions. I create a new exception class CustomerFailedException to distinguish SimPy process-related exceptions from proper Python exceptions.

```
import random
from collections import deque
import simpy
# Shared global variables and "constants"
SERVICE_DELAY = 10
service_line = deque()
counter_idle = False # The state of the clerk at the service counter
env = simpy.Environment()
class CustomerFailedException(Exception):
    pass
```

My model consists of processes of three types. The customer_generator function imitates the rest of the world. In a complete model, the customers would arrive from other model components rather than being "born" on the spot.

```
def customer_generator():
    for _ in range(10):
        env.process(customer())
        yield env.timeout(random.expovariate(1 / SERVICE_DELAY))
customer_generator_p = env.process(customer_generator())
```

Customers in the model are simple stateless creatures. Once born, they obtain a service ticket represented as a simpy.events.Event and enter it into the service_line. If the service counter clerk is asleep (counter_idle==True), the customer interrupts the counter process by calling service_counter_p.interrupt. This method raises a simpy.Interrupt exception in the target process and "awakens" it.

Now that the clerk is not asleep anymore, the customer yields the ticket. The event gets *triggered*. The customer is blocked until some other process processes the event. Naturally, the event becomes *processed*, and if it is processed successfully, the execution of the customer resumes. Otherwise, a CustomerFailedException is raised, which terminates the simulator, if not correctly handled.

(Note: If you are familiar with the C pthreads, you may recognize a striking similarity between SimPy events and pthread_cond_t. Triggering and processing an event are loose equivalents of pthread_cond_wait and pthread_cond_signal, respectively.)

```
def customer():
    print("Customer arrived @{0:.1f}".format(env.now))
    ticket = env.event()
    service_line.append(ticket)
    if counter_idle:
        service_counter_p.interrupt()
    try:
        yield ticket
        print("Customer left @{0:.1f}".format(env.now))
    except CustomerFailedException:
        print("Customer failed (and left) @{0:.1f}".format(env.now))
```

The service counter is a potentially infinite process. If there is at least one waiting customer, the first ticket is removed from the service line, and the corresponding customer is taken care of. Then the counter process "flips a coin" and either fails the ticket event (ticket.fail (CustomerFailedException())) or succeeds it (ticket.succeed). In the former case, a CustomerFailedException is raised at the client side, as explained above.

If the service line is empty, the counter clerk becomes idle and yields an event. This event is triggered, but never processed, putting the clerk into an eternal sleep. However, the sleep can be interrupted by an arriving customer process, also as explained above.

```
def service_counter():
    global counter_idle
    while True:
        if service_line:
            ticket = service_line.popleft()
            yield env.timeout(SERVICE_DELAY)
            if random.randint(0,9) == 9:
                ticket.fail(CustomerFailedException())
            else:
                ticket.succeed()
        else:
            counter_idle = True
            print("The clerk fall asleep @{0:.1f}".format(env.now))
            try:
                yield env.event()
            except simpy.Interrupt:
                counter_idle = False
                print("The clerk woke up @{0:.1f}".format(env.now))
service_counter_p = env.process(service_counter())
```

A sample output of the simulation follows:

```
env.run()
=>The clerk fall asleep @0.0
=>Customer arrived @0.0
=>The clerk woke up @0.0
=>Customer arrived @0.4
=>Customer left @10.0
=>The clerk fall asleep @20.0
=>Customer left @20.0
=>Customer arrived @24.6
=>The clerk woke up @24.6
=>The clerk fall asleep @34.6
=>Customer left @34.6
=>Customer arrived @59.3
=>The clerk woke up @59.3
=>Customer arrived @64.2
=>Customer arrived @65.9
=>Customer left @69.3
=>Customer arrived @75.8
=>Customer left @79.3
=>Customer arrived @83.1
=>Customer arrived @86.7
=>Customer left @89.3
=>Customer arrived @90.9
=>Customer left @99.3
=>Customer left @109.3
=>Customer left @119.3
=>The clerk fall asleep @129.3
=>Customer left @129.3
```

I leave it to you to instrument the model with statistics-gathering tools.

Conclusion

The goal of this article was to demonstrate the possibility of using Python for not-so-trivial M&S projects. I confess that the "M" (modeling) part of the story is somewhat sketchy. In my defense, the field of computer modeling, even just discrete event modeling alone, is too vast to cover in a dozen of pages without distracting the reader's attention from the holistic M&S experience.

On the SimPy side, I went over the most notable features that should be sufficient for many simple M&S projects. Here's a summary of what was left uncovered:

- env.schedule(event, priority=1, delay=0) — A mechanism for scheduling an event in the future, rather than triggering it right away.
- event.callbacks — A list of callback functions that will be called when the event is processed. (The default callback always resumes the blocked process that triggered the event.)
- PriorityResource, PreemptiveResource, PriorityStore, FilterStore — A collection of specialized resources.
- RealtimeEnvironment — An environment for real-time simulation that requires synchronization between the simulation clock and the "wall clock."

I refer you to the SimPy documentation [U6] for the additional information.

About the Author

Dmitry Zinoviev is a professor of Computer Science at Suffolk University, Boston. His research interests include computer modeling and simulation, network analysis, computational social science, and digital humanities. He is the author of *Data Science Essentials in Python* [U7] and *Complex Network Analysis in Python* [U8] published by the Pragmatic Bookshelf.

External resources referenced in this article:

[U1] https://openmodelica.org

[U2] https://ptolemy.eecs.berkeley.edu

[U3] https://www.mathworks.com/products/simulink.html?s_cid=wiki_simulink_2

[U4] http://www.uio.no/om/aktuelt/rektorbloggen/2017/50-years-anniversary-of-simula-the-first-object-or.html

[U5] http://simpy.readthedocs.io/en/latest/

[U6] http://simpy.readthedocs.io/en/latest/topical_guides/

[U7] https://pragprog.com/book/dzpyds/data-science-essentials-in-python

[U8] https://pragprog.com/book/dzcnapy/complex-network-analysis-in-python