Technical University of Munich, Chair of Communication Networks
Prof. Dr.-Ing. W. Kellerer, Răzvan-Mihai Ursu, Polina Kutsevol, Alba Jano

Analysis, Modeling and Simulation of Communication Networks
SS 2024

Programming Assignment 1

Submission Deadline: 7. May 2023

1 Implementing the Discrete Event Simulator (130 Points)

In the first part of the exercise, you will create a Discrete Event Simulator step by step. This section consists of four parts:

- Modeling Server and Queue
- Implementing the Event Chain
- Processing and Generating the Events
- Creating the Simulator

After that, you will verify your system and answer questions on your implementation.

1.1 Modeling Server and Queue

In this part, you will implement the server and the queue. The queue (buffer) stores packets generated by the user that can not be processed by server directly, thus, have to wait for other packets to be served first. This means the server has to finish the operation on the previously arrived packets first. To keep your first implementation simple, you will not have to work on a detailed model for network packets in this assignment.

In your implementation, you are going to have a single user and a single server. For the server, you need an indication (flag) showing whether the server is busy. Moreover, the system contains a counter showing how many packets are currently waiting in the queue. The maximum queue size is limited by the simulation parameter S, the value of which can be set in *simparam.py*. Check what are the other parameters of the simulator in *simparam.py*.

Upon the arrival of a new packet, the system attempts to add it to the service directly bypassing the buffer. If it is not possible because buffer is not empty or server is busy, the packet is added to the buffer, except for the cases when the buffer is full and the packet is dropped.

Now, implement the following tasks in the *SysState* class in the file (Python module) named *systemstate.py* and in the class *Server* in the file *server.py*.

Task 1.1.1: Adding packet to buffer

5 Points

Modify the function *add_packet_to_buffer()* in *systemstate.py*. If it is possible to add new packet to the buffer, the function should modify the variable *buffer_content* accordingly and return *true*. Note that *buffer_content* indicates how many packets are already stored in the buffer.

Task 1.1.2: Initializing and resetting server

5 Points

Initialize the server in the function *__init__()* in *server.py*, assigning it with the simulation object it belongs to and a service rate parameter. The service rate measures how many packets the server can process in one millisecond. In addition, define a boolean attribute *server_busy* that indicates whether the server is currently processing some packet. Initialize *server_busy* with a plausible value. Modify the method *reset()* of the *Server* class such that *server_busy* is reset to initial value and *service_rate* is set through the input of the function *reset()*.

Task 1.1.3: Starting service

10 Points

Modify the function *start_service()* in *server.py*, in which the server attempts to start processing some packet. If the input parameter *pkt_arrived* is set to *true*, that indicates that there is a new packet in the system which can be served bypassing the buffer. If *pkt_arrived* is *false*, the server can only process the packet from the buffer if it is not empty. The function *start_service()* should return *false* if the server could not start processing because it is either busy or there is no packet to serve. Otherwise, if the service can be started, the function should return *true*. Modify the attribute *server_busy* of the server and *buffer_content* of the system state accordingly. Note that the function *start_service()* should not add any packets to the buffer, since this is done in the method *add_packet_to_buffer()* of *SysState* class.

Task 1.1.4: Completing service

5 Points

Modify the attributes of the *Server* within *complete_service()* function in *server.py* that is called when the server finishes processing, i.e., becomes free.

Task 1.1.5: Starting serving new packet

5 Points

Modify the function *start_serving_packet()* in *systemstate.py* that is called upon the arrival of the new packet. It returns *true* if the packet can be served directly bypassing the buffer, i.e., when the buffer is empty and the server can start service.
Hint: Call the corresponding method of the *Server* class with appropriate parameters.

Task 1.1.6: Tracking packets

5 Points

The functions *packet_entered()* and *packet_dropped()* should be called every time new packet enters the system (incl. packets that are going to be blocked) or packet is discarded due to full buffer. Call these functions in the appropriate places within *SysState* class. Note that there is a method *get_blocking_probability()* of *SysState* that can be called to obtain current blocking probability.

1.2 Event Chain

Your DES is based on an event chain that stores all different *SimEvents*. Each *SimEvent* has a time stamp showing the time when it is scheduled, and an order indication to make sure that when multiple events occur at the same time, they will be processed in the correct order.

In this part, you will implement the functionality of the event chain, that is, inserting and removing events in the right and correct manner. For this, you have to modify two classes that you both find in the file *event.py*: *SimEvent* and *EventChain*.

Task 1.2.1: Comparing two events

5 Points

In order to correctly insert a *SimEvent* into the event chain, the events have to be compared with each other. We use the structure *heapq* for the event chain (`https://docs.python.org/3.7/library/heapq.html`). Whenever an event is inserted in our event chain, events in event chain are sorted by their time and order. The earliest events are always processed first. If two events happen at the same time, the one with higher priority, i.e., lower order value, is processed first. Implement the function *__lt__()* according to the above specification. Have a look at the python documentation, how it is used. Hint: this is called operator overloading.

Task 1.2.2: Inserting and removing events

5 Points

Modify the methods *insert()* and *remove_oldest_event()* in the class *EventChain* according to the above functionality descriptions of the event chain. The variable *event_list* should be modified using methods of *heapq*, whenever one of these two functions is called. The function *remove_oldest_event()* should return the event of interest.

1.3 Processing and Generation of Events

In the given DES, there are three types of events: Packet Arrival, Service Completion and Simulation Termination. In this part, you are going to implement the processing functions of all event types in *event.py*, as well as the functions for generating new events by server and user in *server.py* and *user.py*.

Task 1.3.1: Generating packet arrival

5 Points

Implement method *generate_packet_arrival()* of *User* class in *user.py*. This function is going to be called within processing of the packet arrival event to generate the next arrival. Inter-arrival time of two consecutive packets is random and follows uniform distribution with the mean corresponding to service rate of the user. In the function *generate_packet_arrival()*, you should first generate a uniformly distributed inter-arrival time using the function *random.uniform()* (`https://docs.python.org/3.7/library/random.html`). It should draw a time in milliseconds between 1 and $2 * mean\_iat$, where *mean_iat* corresponds to the arrival rate. Store the generated number in *last_iat* attribute. Then, you should create a *PacketArrival* event with the timestamp that is *last_iat* later from the current time. Finally, you should insert the event to the event chain of the simulation. Note that the current simulation time is stored in *sys_state* object that you can access through the *sim* attribute of *User*. Hint: check the initialization of *PacketArrival* class to correctly generate the *PacketArrival* object.

Task 1.3.2: Packet arrival processing

10 Points

This event should occur once a new packet arrives in your system. In details, implement function *process()* of *PacketArrival* class in *event.py* that should make the system either serve packet directly, enqueue or drop it, depending on the current system state. For that, use the functions of *SysState* class that you implemented before. Make sure that each *PacketArrival* event inserts a new *PacketArrival* into the event chain.

Task 1.3.3: Generating service completion

5 Points

Implement method *generate_service_completion()* of *Server* class in *server.py*. This function should be called once the server starts processing some packet and generate an event of finishing the process. Processing time of the service is fixed and corresponds to the service rate of the server. Within the function *generate_service_completion()*, you should create a *ServiceCompletion* event with the timestamp that is later from the current time by the processing time. Also, you should insert the event to the event chain of the simulation. Call the function *generate_service_completion()* in the appropriate place within *server.py*.
Hint: check the declaration of *ServiceCompletion* class to correctly generate the *ServiceCompletion* object.

Task 1.3.4: Service completion processing

10 Points

This event should occur once the server completes the processing. Implement a function *process()* of *ServiceCompletion* in *event.py* that completes serving the current packet and starts processing the next packet from buffer if possible. Use the functions of *Server* class that you implemented before.

Task 1.3.5: Simulation termination

5 Points

This event is only processed at the end of a simulation run. Adapt the *process()* method of *SimulationTermination* class to make sure that the simulation stops when this event occurs. The function should modify the attributes in class *SysState* for this.

1.4 Creating the Simulator

Now you are ready to setup the whole simulator and make a simulation run. For this, the function *do_simulation()* in the python file *simulation.py* has to be modified. You see that the stream of the packet arrivals is initiated with *start_users()* function, and the last event of your simulation is inserted. All other events are inserted on the fly according to the implementation. The basic structure contains a while loop that runs until the *sys_state.stop* flag is set. In each iteration of the while loop, only one event is processed and the simulation time (*sys_state.now*) is updated accordingly.

Task 1.4.1: Creating the Simulator

10 Points

Modify the above-mentioned function *do_simulation()* such that the simulator processes the events in the right and correct manner. Pay attention here and think carefully about how and when to update the simulation time, since this can strongly affect the way your simulation works.

1.5 Verification

Since you have setup the simulator, it's time to verify your simulation results. We provide you a python unittest file that checks the basic functionality of your code. The unittests in file *test1* check the functionalities of different parts of simulaton. Furthermore, it can run whole simulations with the seed ranging from 0 to 5, helping you to check the correct implementation of your DES. Note that for the tests to pass, you should leave the default parameters of the simulation in *simparam.py*, i.e., maximum queue size is 4 ($S = 4$), the simulation time is 100s (100000ms), and arrival and service rates are 0.002 and 0.0015.

When you run a simulation, the system should count roughly between 35 and 65 dropped packets. Try different seeds to check whether this is the case in your simulation. Run all unittests and make sure they do not return any errors.

1.6 Analysis and General Questions

The last section of the first programming assignment contains a few questions that should be answered separately. Explain your answers and write full sentences.

Task 1.6.1: Event-based Simulator

5 Points

Why is it feasible to use Next-Event Time Advance simulator and not Fixed-Increment Time Advance simulator for the considered system? Give at least one example when utilizing Next-Event Time Advance would not be efficient or possible?

Task 1.6.2: Event Chain Structure

5 Points

Explain why it makes sense to use a heap as your data structure for the event chain instead of other data structures.

Task 1.6.3: Emergency Termination

5 Points

In your implementation, the simulation is terminated upon the processing of the *Simulation-Termination* event according to its timestamp. Imagine you would like to include another event called *EmergencyTermination* defined as follows:

```
class EmergencyTermination(SimEvent):
    """
    Defines the emergency termination of a simulation.
    """

    def __init__(self, sim, timestamp):
        """
        Create an emergency simulation termination event
        independent of the execution time.

        Order of an emergency simulation termination
        event is set to the flag "Emergency"
        """
        super(EmergencyTermination, self).__init__(sim, timestamp)
        self.order = "Emergency"
```

Upon the insertion of the *EmergencyTermination* event to the event chain, the simulation should stop immediately independent of *EmergencyTermination* timestamp and the current simulation time. Using the *"Emergency"* flag set to the *order* attribute, how would you modify the *__lt__()* method of *SimEvent*, such that the *EventChain* processes the *EmergencyTermination* event according to the specified functionality?

Note: do not change the functions in DES code but present the required implementation as a separate function in the Jupyter Notebook with your answers.

1.7 Simulation Study I

In the previous tasks, you have created a basic version of a DES. In this section perform a simulation study. Create a new file *simstudy1.py* and write your code in this file. Explain your answers and write full sentences.

Task 1.7.1: Utilization Determination I

10 Points

Find out what is the maximum utilization $\rho = \frac{\lambda}{\mu}$ that the system with $S = 4$ can handle such that after 100 s, less than 5 packets are lost in at least 95% of 1000 simulation runs. Keep the service rate default, i.e., $\mu = 0.0015$. and vary arrival rate λ. Determine ρ to the precision of two decimal places (i.e., in the format x.xx). Describe your idea how to design the simulation runs to achieve this goal. Think about cases, when you are close to fulfilling or not fulfilling the requirements and how you could assure a correct result in these cases.

Task 1.7.2: Offered Traffic Determination II

5 Points

Which ρ will you get if you change μ to 0.015 and want to have less than 50 dropped packets in 95% of the cases? Describe your observations!

Task 1.7.3: Comparison of Results

10 Points

Compare the results of task 1.7.1 and task 1.7.2. Does the system behave differently or not? Explain your observations! Hint: Think about the distribution of the blocking probability per run. Possible solution: Plotting cumulative distribution functions of the blocking probabilities might be helpful.

Total: 130 Points