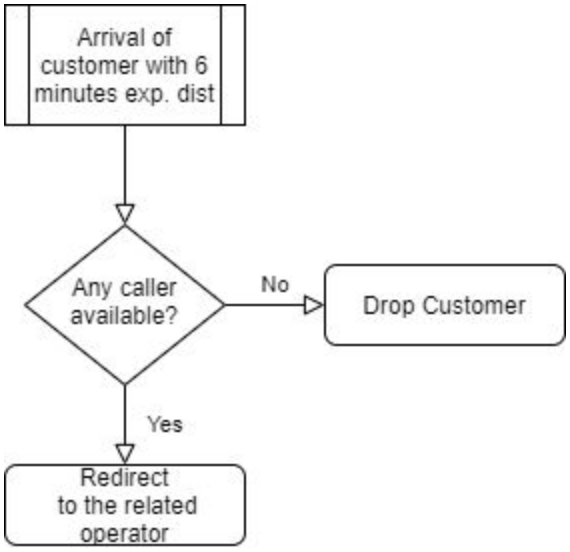
IE.306 System Simulation

Assignment-1

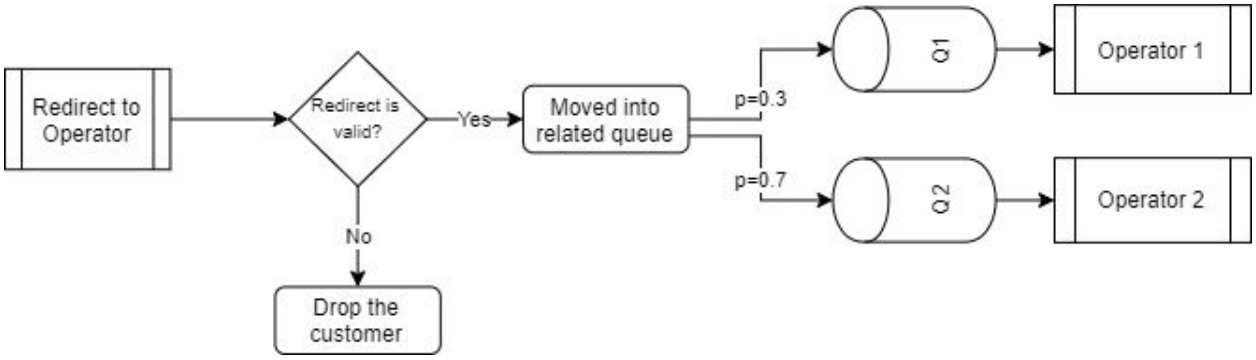
2009400189-2015400132- 2015400183

1) Description the Problem

System consists of two main parts as callers processing and redirecting the customers and operators serving the customers.



The logic of the first part is described above. There are 100 callers available and if there is none available for customers at the time of arrival, the customer will be dropped immediately. The process of identifying and redirecting is exponential distribution with 5 mins mean. Redirect process sometimes makes an error with probability 0.1 and sends it to the wrong operator.



Second part is serving customers which starts with moving the customer to the related queue. If the queue is empty then the customer is served right away. A simple representation shown above.

2) Definitions of the Model

System state variables for storing the all related variables and their variables in the python code declared as:

- Service times of customers on callers => service_times_aam **= []**
- Service times for operator one => service_times_operator_1 **= []**
- Service times for operator two => service_times_operator_2 **= []**
- Waiting times in the queue for operator one => queue_waiting_times_operator_1 **= []**
- Waiting times in the queue for operator two => queue_waiting_times_operator_2 **= []**
- Unsatisfied customers because of wrong routing => unsatisfied_customers_incorrect_routed **= []**
- Unsatisfied customers because of waiting too long => unsatisfied_customers_overwaiting **= []**

There is one main entity defined in the system which is the **Customer** itself. Attributes of a customer are:

- arrival time
- time of leaving
- and operator to be served on.

Events thats changes the state of the system:

- Arrival of a customer into callers
- End of caller processing
- Dropping customer due to wrong routing
- Arrival of a customer into operators
- Dropping of a customer after waiting ten minutes
- Finishing serving to a customer
- Operator taking a break

Implementation

Simulation is implemented on simpy utilizing the JupyterLab. `simulaton.py` can be opened in a Jupyter notebook and running it will write the outputs to the console. Additional libraries needed to be installed:

- simpy using `pip install simpy`
- numpy using `pip install numpy`

All the variables required for defining probability distributions for events initialized at the start.

```
# Distribution related parameters initialized.
INTERARRIVAL_MEAN = 6.0
INTERARRIVAL_RATE = 1.0 / INTERARRIVAL_MEAN
AAM_SERVICE_TIME_MEAN = 5.0
AAM_SERVICE_TIME_RATE = 1.0 / AAM_SERVICE_TIME_MEAN
OPERATOR_1_SERVICE_TIME_MEAN = 12.0
OPERATOR_1_SERVICE_TIME_STDEV = 6.0
QUEUE_WAITING_TIME_LIMIT = 10.0
BREAK_TIME = 3
BREAK_TIME_RATE = 1/60.0
```

```
CUSTOMERS_SERVICED = 0
```

After that system state variables are defined which stores customer and operator related data. Both the caller system and operators have different arrays used for service times.

```
service_times_aam = []

service_times_operator_1 = []

service_times_operator_2 = []
```

Operators have different waiting queues and unsatisfied customers from callers and operators are stored separately.

```
queue_waiting_times_operator_1 = []

queue_waiting_times_operator_2 = []

unsatisfied_customers_incorrect_routed = []
unsatisfied_customers_overwaiting = []
```

The main entity in the system is the customer. Every customer is defined with a name, arrival time, initial caller action, operator to be forwarded, and stopping event. Actions follow the flow logic described above. We will chain the actions as caller, forwarding, operator queue, and serving.

```
#Arriving customers are initialized.
class Customer(object):
    def __init__(self, name, env, opr_num, sample_size, stopping_event):
        self.env = env
        self.name = name
        self.arrival_t = self.env.now
        self.action = env.process(self.call())
        self.opr_num = opr_num
        self.sample_size = sample_size
        self.stopping_event = stopping_event
```

After a customer is created it will run the `call` function to process the caller action. If all callers are busy then the call will be dropped immediately. Otherwise a random service time will be assigned and added to the service array.

```
    #Customers call automated answering mechanism.
    def call(self):
        global CUSTOMERS_SERVICED
        with automated_answer_mech.request() as req:
            # If the automated answer mechanism is full, the customer is dropped from the system.
            if automated_answer_mech.count == automated_answer_mech.capacity:
                return
            aam_service_duration = random.expovariate(AAM_SERVICE_TIME_RATE)
```

```
            yield self.env.timeout(aam_service_duration)
            automated_answer_mech.release(req)
            service_times_aam.append(aam_service_duration)
```

After the caller identifies the customer and its operator we create a uniform random variable for routing. Having the random lower than 0.1 means there is an error and the customer is routed to the wrong operator. In this case this customer is added to the unsatisfied customer list from routing.

```
        # The misrouted customers with a probability of 0.1 are dropped.
        incorrect_routing_probability = random.uniform(0, 1)
        if incorrect_routing_probability < 0.1:
            unsatisfied_customers_incorrect_routed.append(self.name)
            return
```

If the routing was successful the customer will be redirected to the related operator by calling respective functions.

```
        # Correctly routed customers are redirected into the operators.
        if self.opr_num == 1:
            yield self.env.process(self.call_operator_1())
        else:
            yield self.env.process(self.call_operator_2())
```

Even though operator functions are quite similar we kept them separate to make it easier to read. Operator drops the customers which are waiting for too long from the queue before serving a new one. Queue waiting is defined as a timeout that triggers an event (as not triggering serving) after ten minutes. Any customer who reaches the timeout will be dropped and added to unsatisfied customer list.

```
    # The customers that want to use operator 1 are connected.
    def call_operator_1(self):
        global CUSTOMERS_SERVICED
        queue_start_time = self.env.now
        with operator_1.request(priority = 0) as req:
            yield req | self.env.timeout(QUEUE_WAITING_TIME_LIMIT)
            # The customers that wait more than 10 minutes are dropped.
            if not req.triggered:
                queue_waiting_times_operator_1.append(QUEUE_WAITING_TIME_LIMIT)
                unsatisfied_customers_overwaiting.append(self.name)
                return
```

When a customer comes to the operator from the queue, a uniformly distributed random variable is created for service time. It is also added to the service time list of the operator. After the serving time is completed we increase the number of customers served and check if the total number reached the sample size. In that case the simulation is completed and the stopping event will be triggered.

```
queue_waiting_time = self.env.now - queue_start_time
if queue_waiting_time != 0:
    queue_waiting_times_operator_2.append(queue_waiting_time)

operator_2_service_duration = random.uniform(1,7)
yield self.env.timeout(operator_2_service_duration)
service_times_operator_2.append(operator_2_service_duration)
CUSTOMERS_SERVICED += 1
# If the customer serviced is the last customer, then the
# stopping event that terminates the simulation is triggered.
if CUSTOMERS_SERVICED == self.sample_size:
    self.stopping_event.succeed()
```

Customers are generated with interarrival times for the caller service. After creating a caller serving time we assign the operator depending on the uniformly distributed random variable.

```
def customer_generator(env, sample_size, stopping_event):
    while True:
        yield env.timeout(random.expovariate(INTERARRIVAL_RATE))
        probability = random.uniform(0,1)
        opr_num = 1 if probability <= 0.3 else 2
        customer = Customer('Cust %s' %(i+1), env, opr_num, sample_size, stopping_event)
```

For the operators break times generated by a poisson distribution. Priority is given as one for breaks and zero for customer serving events. This way we guarantee that all the customers will be served before operators taking a break since they have a higher priority.

```
def break_call(env, operator):
    while True:
        yield env.timeout(np.random.poisson(BREAK_TIME_RATE))
        with operator.request(priority = 1) as req:
            yield req
            yield env.timeout(BREAK_TIME)
```

We used ten different seeds and two different customer numbers as 1000 and 5000. For each seed value a new random is created to be used in the simulation.

```
SEED_LIST = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
CUSTOMER_SIZE_LIST = [1000, 5000]

for seed in SEED_LIST:
    random.seed(seed)
    np.random.seed(seed)
    for (i, current_size) in enumerate(CUSTOMER_SIZE_LIST):
```

Using the simpy environment we defined the stopping event, resources for callers and operators. Default provided capacity parameters are used as 100 for callers and 1 for operators. At last we define the processes defined above as parameters to the environment for actions and run it until the stopping event.

```
env = simpy.Environment()
stopping_event = env.event()
automated_answer_mech = simpy.Resource(env, capacity=100)
operator_1 = simpy.PriorityResource(env, capacity=1)
operator_2 = simpy.PriorityResource(env, capacity=1)
env.process(customer_generator(env, current_size, stopping_event))
env.process(break_call(env, operator_1))
env.process(break_call(env, operator_2))
env.run(until=stopping_event)
system_time = env.now
```

The final part is collecting statistics from the data we collected in the simulation. Every variable is appended to the corresponding lists and we will take the average values from these lists while calculating the outputs.

Simulation and Results

Simulation is completed with two different sample sizes and ten different seeds. Sample sizes are 1000 and 5000 customers respectively. Outputs of simulation are given below.

Simulation of the System for **1000** Customers:

Utilization of Answering System: **0.008168507423859937**

Utilization of Operator 1: **0.44230979895347133**

Utilization of Operator 2: **0.40536237756685145**

Average Total Waiting Time: **3.5782916099952558**

Maximum Total Waiting Time to Total System Time Ratio: **0.5498132061287802**

Average Number of People Waiting to be Served by Operator 1: **328.3**

Average Number of People Waiting to be Served by Operator 2: **753.5**

Average Number of Unsatisfied Customers: **209.4**

Simulation of the System for **5000** Customers:

Utilization of Answering System: **0.008356072314241769**

Utilization of Operator 1: **0.44620337247506814**

Utilization of Operator 2: **0.40829447705081223**

Average Total Waiting Time: **3.6350333551253002**

Maximum Total Waiting Time to Total System Time Ratio: **0.5031171573551524**

Average Number of People Waiting to be Served by Operator 1: **1647.2**

Average Number of People Waiting to be Served by Operator 2: **3777.9**

Average Number of Unsatisfied Customers: **1040.1**

The collection of statistics leads us into the derivation of results by observing the data. For both sizes, we observe that the utilization of system servers are nearly the same. The average waiting times also remain unchanged, except a minor decrease in maximum total waiting time to total system time ratio. Plus, we see that there exists a linear relationship by the factor of 5 between the system size and the average number of people waiting to be served and unsatisfied customers. We can conclude that as the system size grows, the long-run measures related to taking the average show a linear growth as well, while utilization measures remain unchanged. This implies that the system size has nearly no effect in system performance.