```python
import heapq
import random
import matplotlib.pyplot as plt
import numpy as np
from scipy.stats import t

"""
Class to characterize an event.

Attributes:
    time: at which time the event occurs
    event_type: 'arrival' or 'departure' (also 'end' to manage the end of a simulation)
"""
class Event:
    def __init__(self, time, event_type):
        self.time = time
        self.event_type = event_type

    def __lt__(self, other):
        return self.time < other.time

"""
Class to perform the simulation.

Attributes:
    arrival_rate: λ
    service_rate: μ
    simulation_time: for how long we make the simulation last (last arrival is scheduled before simulation_time, then we manage the last departures)

    event_queue: store the events scheduled (it's a priority queue where priority is based on time and we use a binary heap to manage it)
    current_time: current time of the simulation
    server_busy: True or False to determine whether is the server is currently busy or not
    queue_length: we save the number of packets in queue

    packets_in_system: #packets in the system at a given time (used to compute average number of packets in the system)
    packets_in_queue: #packets in the queue at a given time (used to compute average time in the queue)
    arrival_times_queue: it represents the state of the system (which packets there are in the queue)
    time_in_system: it stores, for each packet, how much time it stayed in the system and how many packets it had in front when entered the system
"""
class MM1QueueSimulator:

    def __init__(self, arrival_rate, service_rate, simulation_time):
        self.arrival_rate = arrival_rate
        self.service_rate = service_rate
        self.simulation_time = simulation_time

        self.event_queue = []
        self.current_time = 0
        self.server_busy = False
        self.queue_length = 0

        self.packets_in_system = []
        self.packets_in_queue = []

        self.arrival_times_queue = []
        self.time_in_system = []

    """
    Add an event in queue.
    """
    def schedule_event(self, event):
        heapq.heappush(self.event_queue, event)

    """
    Draw from an exponential
    """
    def generate_time(self, rate):
        return random.expovariate(rate)

    """
    Method to handle the simulaton
    """
    def simulate(self):
        #schedule the first arrival and simulation end
        self.schedule_event(Event(self.generate_time(self.arrival_rate), "arrival"))
        self.schedule_event(Event(self.simulation_time, "end"))

        #print(f"Simulation started at time {self.current_time}")
        while self.event_queue:

            event = heapq.heappop(self.event_queue)
            self.current_time = event.time

            if event.event_type == "arrival":
                #print(f"Arrival at time {self.current_time} [current in system {self.queue_length + int(self.server_busy)}]")
                self.handle_arrival()
            elif event.event_type == "departure":
                #print(f"Departure at time {self.current_time} [current in system {self.queue_length + int(self.server_busy)}]")
                self.handle_departure()
            elif event.event_type == "end":
                #print(f"Simulation ended at time {self.current_time} [current in system {self.queue_length + int(self.server_busy)}]. I still could need to ha
                break

            self.packets_in_system.append((self.current_time, self.queue_length + int(self.server_busy)))
            self.packets_in_queue.append((self.current_time, self.queue_length))

        #After the end event, we will have at most some departures
        #Process last departures in the queue
        while self.event_queue:

            event = heapq.heappop(self.event_queue)
            self.current_time = event.time

            if event.event_type == "arrival":
                print(f"NEVER HAPPENS")
                self.handle_arrival()
```

```python
            elif event.event_type == "departure":
                #print(f"Departure at time {self.current_time} [current in system {self.queue_length + int(self.server_busy)}]")
                self.handle_departure()
            elif event.event_type == "end":
                print(f"NEVER HAPPENS")

            self.packets_in_system.append((self.current_time, self.queue_length + int(self.server_busy)))
            self.packets_in_queue.append((self.current_time, self.queue_length))

    """
    Method to handle an arrival event
    """
    def handle_arrival(self):

        actual_queue_length = self.queue_length
        in_server = 1 if self.server_busy else 0

        if self.server_busy:
            self.queue_length += 1
        else:
            self.server_busy = True
            self.schedule_event(Event(self.current_time + self.generate_time(self.service_rate), "departure"))

        #we store also how many packets it has in front when entered the system
        self.arrival_times_queue.append((self.current_time, actual_queue_length + in_server))

        #schedule the next arrival - only if it can arrive until the end of the simualtion
        next_arrival = self.current_time + self.generate_time(self.arrival_rate)
        if next_arrival < self.simulation_time:
            self.schedule_event(Event(next_arrival, "arrival"))

    """
    Method to handle a departure event
    """
    def handle_departure(self):

        #We take the arrival time of the actual packet we are processing in order to compute how much time it spent in the system
        #We store also how many packets it got in front when it arrived
        arrival_time, packets_in_front = self.arrival_times_queue.pop(0)
        time_in_system = self.current_time - arrival_time
        self.time_in_system.append((time_in_system, packets_in_front, arrival_time))

        if self.queue_length > 0:
            self.queue_length -= 1
            self.schedule_event(Event(self.current_time + self.generate_time(self.service_rate), "departure"))
        else:
            self.server_busy = False

    ############STATISTIC COMPUTATION############
    """
    Method to compute empirically the average number of packets in the system (we consider only the events happening after warmup_time)
    """
    def average_packets_in_system(self, warmup_time=0.0):

        if warmup_time != 0.0:
            start_index = 0
            for i, (t, _) in enumerate(self.packets_in_system):
                if t >= warmup_time:
                    start_index = i
                    break
            filtered_data = self.packets_in_system[start_index:] #no need to zero the time to the first event we consider since we are interested in deltas
        else:
            filtered_data = self.packets_in_system


        total_time = 0
        weighted_sum = 0

        for i in range(len(filtered_data) - 1):
            time_current, packets_current = filtered_data[i]
            time_next, _ = filtered_data[i + 1]

            #time interval
            delta_time = time_next - time_current

            #add to weighted sum
            weighted_sum += packets_current * delta_time

            total_time += delta_time

        average_packets = weighted_sum / total_time
        return average_packets

    """
    Method to compute empirically the average number of packets in the queue (we consider only the events happening after warmup_time)
    """
    def average_packets_in_queue(self, warmup_time=0.0):

        if warmup_time != 0.0:
            start_index = 0
            for i, (t, _) in enumerate(self.packets_in_queue):
                if t >= warmup_time:
                    start_index = i
                    break
            filtered_data = self.packets_in_queue[start_index:]
        else:
            filtered_data = self.packets_in_queue

        total_time = 0
        weighted_sum = 0

        for i in range(len(filtered_data) - 1):
            time_current, packets_current = filtered_data[i]
            time_next, _ = filtered_data[i + 1]

            #time interval
```

```python
            delta_time = time_next - time_current

            #add to weighted sum
            weighted_sum += packets_current * delta_time

            total_time += delta_time

        average_packets = weighted_sum / total_time
        return average_packets

    ############PLOTTING#############

    """
    We plot #packets in the system vs time
    """
    def plot_packets_in_system(self, warmup_time=0.0):
        rho = self.arrival_rate / self.service_rate
        theoretical_value = rho / (1 - rho)
        empirical_value = self.average_packets_in_system(warmup_time)

        if warmup_time != 0.0:
            start_index = 0
            for i, (t, _) in enumerate(self.packets_in_system):
                if t >= warmup_time:
                    start_index = i
                    break
            filtered_data = self.packets_in_system[start_index:]
        else:
            filtered_data = self.packets_in_system

        times, packets = zip(*filtered_data)
        plt.plot(times, packets, drawstyle='steps-post', color="red")
        plt.xlabel("Time")
        plt.ylabel("Number of Packets in System")
        plt.title("Number of Packets in System Over Time")
        plt.text(
            0.02, 0.90,
            f"Theoretical value: {theoretical_value:.3f}\nEmpirical value: {empirical_value:.3f}",
            transform=plt.gca().transAxes,
            color='black',
            bbox=dict(facecolor='white', edgecolor='black', boxstyle='round,pad=0.3')
        )
        plt.show()

    """
    We plot average #packets in the system vs time
    """
    def plot_cumulative_packets_in_system(self, warmup_time=0.0):

        rho = self.arrival_rate / self.service_rate
        theoretical_value = rho / (1 - rho)

        if warmup_time != 0.0:
            start_index = 0
            for i, (t, _) in enumerate(self.packets_in_system):
                if t >= warmup_time:
                    start_index = i
                    break
            filtered_data = self.packets_in_system[start_index:]
        else:
            filtered_data = self.packets_in_system

        cumulative_time = []
        cumulative_avg_packets = []

        total_time = 0
        total_weighted_packets = 0

        for i in range(len(filtered_data) - 1):
            t_curr, n_curr = filtered_data[i]
            t_next, _ = filtered_data[i + 1]
            dt = t_next - t_curr

            total_time += dt
            total_weighted_packets += n_curr * dt

            cumulative_time.append(t_next)
            cumulative_avg_packets.append(total_weighted_packets / total_time)

        plt.plot(cumulative_time, cumulative_avg_packets, label=f"Empirical {self.average_packets_in_system():.3f} ({warmup_time} warmup)" if warmup_time > 0 e
        plt.axhline(theoretical_value, color='black', linestyle='--', label=f'Expected: {theoretical_value})')
        plt.xlabel("Time")
        plt.ylabel("Average Number of Packets in System")
        plt.legend(loc="best")
        plt.title("Cumulative average number of packets in the system over time")
        plt.show()

        #at the beginning a lot of time in the system then we stabilize to theoretical values

    """
    Here we plot both the metrics
    """
    def plot_together(self, warmup_time=0.0):
        rho = self.arrival_rate / self.service_rate
        theoretical_value = rho / (1 - rho)
        empirical_value = self.average_packets_in_system(warmup_time)

        if warmup_time != 0.0:
            start_index = 0
            for i, (t, _) in enumerate(self.packets_in_system):
                if t >= warmup_time:
                    start_index = i
                    break
            filtered_data = self.packets_in_system[start_index:]
        else:
            filtered_data = self.packets_in_system
```

```python
        times, packets = zip(*filtered_data)

        #Cumulative metric

        cumulative_time = []
        cumulative_avg_packets = []

        total_time = 0
        total_weighted_packets = 0

        for i in range(len(filtered_data) - 1):
            t_curr, n_curr = filtered_data[i]
            t_next, _ = filtered_data[i + 1]
            dt = t_next - t_curr

            total_time += dt
            total_weighted_packets += n_curr * dt

            cumulative_time.append(t_next)
            cumulative_avg_packets.append(total_weighted_packets / total_time)

        fig, axes = plt.subplots(1, 2, figsize=(12, 6))
        ax1 = axes[0]
        ax2 = axes[1]

        ax1.plot(times, packets, drawstyle='steps-post', color="red")
        ax1.set_xlabel("Time")
        ax1.set_ylabel("Number of Packets in System")
        ax1.set_title("Number of Packets in System Over Time")
        ax1.text(
            0.02, 0.50,
            f"Theoretical value: {theoretical_value:.3f}\nEmpirical value: {empirical_value:.3f}",
            transform=plt.gca().transAxes,
            color='black',
            bbox=dict(facecolor='white', edgecolor='black', boxstyle='round,pad=0.3')
        )

        ax2.plot(cumulative_time, cumulative_avg_packets, label=f"Empirical {empirical_value:.3f} ({warmup_time} warmup)" if warmup_time > 0 else f"Empirical
        ax2.axhline(theoretical_value, color='black', linestyle='--', label=f'Expected: {theoretical_value}')
        ax2.set_xlabel("Time")
        ax2.set_ylabel("Average Number of Packets in System")
        ax2.legend(loc="best")
        ax2.set_title("Cumulative average number of packets in the system over time")

        plt.show()

    ############TIME SPENT IN THE SYSTEM#############
    """
    Method to compute empirically the average time spent in the system
    """
    def average_time_in_system(self, warmup_time):
        if warmup_time != 0.0:
            filtered_data = [el[0] for el in self.time_in_system if el[2] >= warmup_time]
        else:
            filtered_data = [el[0] for el in self.time_in_system]

        return np.mean(filtered_data)

    """
    A getter for the time spent in the system of all the packets
    """
    def times_in_system(self):
        return [el[0] for el in self.time_in_system]

    def plot_cumulative_times_in_system(self, warmup_time=0.0):
        if warmup_time != 0.0:
            filtered_data = [el[0] for el in self.time_in_system if el[2] >= warmup_time]
        else:
            filtered_data = [el[0] for el in self.time_in_system]

        running_total = 0
        running_average = []

        for i, t in enumerate(filtered_data):
            running_total += t
            running_average.append(running_total / (i+1))

        plt.plot(running_average)
        plt.axhline(1 / (self.service_rate - self.arrival_rate), color='black', linestyle='--', label='Expected: 1/(μ - λ)')
        plt.xlabel("# packets")
        plt.ylabel("Time")
        plt.legend(loc="best")
        plt.title("Average time in the system after x packet")
        plt.show()


    def plot_times_in_system(self, warmup_time=0.0):
        times = self.times_in_system()
        if warmup_time != 0.0:
            start_index = 0
            for i, t in enumerate(times):
                if t >= warmup_time:
                    start_index = i
                    break
            filtered_data = times[start_index:] #no need to zero the time to the first event we consider since we are interested in deltas
        else:
            filtered_data = times

        plt.plot(filtered_data, alpha=0.7)
        plt.axhline(1 / (self.service_rate - self.arrival_rate), color='black', linestyle='--', label='Expected: 1/(μ - λ)')
        plt.xlabel("Packet")
        plt.ylabel("Time")
        plt.ylim((0.0, max(max(times), max(times)*1.1))
        plt.legend(loc="best")
        plt.title("Time in the system by each packet")
```

```python
        plt.show()



"""
A function to compute the CI doing independent replications

Parameters:
    data: vector of metric computed, once for each replication
    confidence_level: the level of confidence to compute

Returns:
    lower bound and upper bound of the confidence interval
"""
def confidence_interval(data, confidence_level=0.95):
    grand_mean = np.mean(data)
    replications = len(data)
    variance_estimator = sum([ ((data[i] - grand_mean)**2) for i in range(replications)])
    variance_estimator = variance_estimator / (replications - 1)

    degrees_of_freedom = replications - 1
    alpha = (1 + confidence_level) / 2
    t_quantile = t.ppf(alpha, degrees_of_freedom)
    lower_bound = grand_mean - t_quantile * np.sqrt(variance_estimator / replications)
    upper_bound = grand_mean + t_quantile * np.sqrt(variance_estimator / replications)
    return lower_bound, upper_bound
```