



## پروژه‌ی درس آمار و احتمال مهندسی

آشنایی با پردازش سیستم‌های لبه‌ای

استاد درس: دکتر محمد مهدی مجاهدیان

فرنوش چوگانی ۴۰۲۱۰۰۶۹۱

محمد رضا آجورلو ۴۰۲۱۰۱۱۵۸

دانشکده‌ی مهندسی برق

دانشگاه صنعتی شریف

بهمن ماه ۱۴۰۳

---

## پاسخ پرسش شبیه سازی ۱:

ابتدا رفتار سرور لبه را مدلسازی می کنیم. می دانیم کارها از توزیع تصادفی پواسون با نرخ  $\lambda$  تولید می شوند که  $\lambda$  نشان دهنده میانگین تعداد کارهایی است که در واحد زمان به سیستم وارد می شوند. و مدت زمانی که سرور کار را به انجام می رساند توزیع تصادفی نمایی با نرخ  $\mu$  است که  $\mu$  نرخ خدمت دهی متوسط است.

رابطه بین نرخ خدمت دهی و میانگین زمان خدمت دهی به صورت  $E[s] = \frac{\lambda}{\mu} = \rho$ . برای اطمینان از اینکه صفات محدود باشند باید  $\mu < \lambda$  باشد. شرط دور ریخته شدن کاری این است که صفات به حداقل ظرفیت  $K$  خود برسد در نتیجه کارهای ورودی کنار گذاشته می شوند. میانگین تعداد کارهای موجود در صفات برابر است با :

$$L_q = \frac{\rho(1 - P_K)}{1 - \rho} \quad \text{احتمال پر بودن صفات} \quad P_K$$

$$P_K = \frac{\frac{(\lambda/\mu)^K}{K!}}{\sum_{n=0}^K \frac{(\lambda/\mu)^n}{n!}} \quad \text{احتمال پر بودن صفات}$$

میانگین تعداد کارهایی که به دلیل نبود ظرفیت در صفات سرور لبه دور ریخته می شوند برابر است با:

$$R_{drop} = \lambda \cdot P_K \quad \text{نرخ کارهای دور ریخته شده}$$

$$E[droppedworks] = R_{drop} \cdot T = \lambda \cdot P_K \cdot T$$

برای رسم نمودارهای خواسته شده، از کد زیر استفاده می کنیم که بازه  $\lambda$  با توجه به فرض سوال تعیین شده و مقدار  $\mu$  به دلخواه انتخاب شده است. برای رسیدن به حالت ایستا تعداد داده های ارسالی از سنسورها ۱۰۰۰۰۰ انتخاب شده است.

```

import numpy as np
import matplotlib.pyplot as plt

mu = 1
lambda_range = np.linspace(0.05 * mu, 2 * mu, 5)
capacity = 10
time_end = 100
N = 1000
dt = 10
time_steps = int(time_end / dt)
transient_period = int(0.1 * time_steps)

all_times = np.arange(0, time_end, dt)
queue_lengths = []
dropped_jobs = []

for lam in lambda_range:
    queue_length_avg = np.zeros(time_steps)
    dropped_jobs_avg = np.zeros(time_steps)

    for _ in range(N):
        queue = 0
        jobs_dropped = np.zeros(time_steps)
        queue_length_history = np.zeros(time_steps)

        for t in range(time_steps):
            arrivals = np.random.poisson(lam * dt)
            processing = np.random.poisson(mu * dt) if queue > 0 else 0

            queue += arrivals
            if queue > capacity:
                jobs_dropped[t] = queue - capacity
                queue = capacity

            queue -= processing
            queue = max(queue, 0)
            queue_length_history[t] = queue

        queue_length_avg += queue_length_history
        dropped_jobs_avg += jobs_dropped

    queue_length_avg /= N
    dropped_jobs_avg /= N

    dropped_jobs.append(dropped_jobs_avg[transient_period:])
    queue_lengths.append(queue_length_avg[transient_period:])

all_times_trimmed = all_times[transient_period:]

fig, axes = plt.subplots(2, 1, figsize=(10, 8), sharex=True)

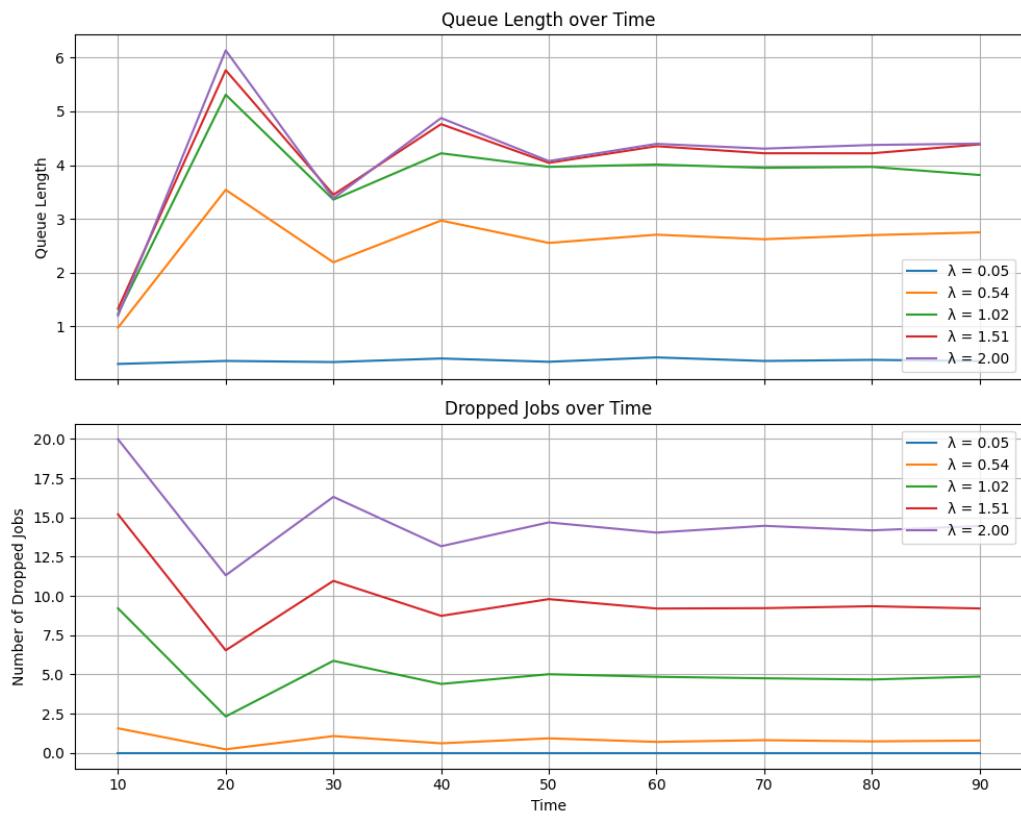
for i, lam in enumerate(lambda_range):
    axes[0].plot(all_times_trimmed, queue_lengths[i], label=f" $\lambda = {lam:.2f}$ ")
    axes[0].set_ylabel("Queue Length")
    axes[0].set_title("Queue Length over Time")
    axes[0].grid()
    axes[0].legend()

    for i, lam in enumerate(lambda_range):
        axes[1].plot(all_times_trimmed, dropped_jobs[i], label=f" $\lambda = {lam:.2f}$ ")
        axes[1].set_xlabel("Time")
        axes[1].set_ylabel("Number of Dropped Jobs")
        axes[1].set_title("Dropped Jobs over Time")
        axes[1].grid()
        axes[1].legend()

plt.tight_layout()
plt.show()

```

خروجی کد به صورت زیر خواهد بود:



برای  $\lambda = 54\%$  مقدار کارهای دور ریخته شده را با فرمول های تئوری محاسبه کردیم و تقریباً به نتایج حاصل از شبیه سازی نزدیک بود.

## پاسخ پرسش شبیه سازی ۲:

```
import numpy as np
import matplotlib.pyplot as plt

def simulate_main_server(mu_main, time_steps, main_queue, collector_queues):
    main_lengths = np.zeros(time_steps)

    for t in range(time_steps):
        for collector_queue in collector_queues:
            while collector_queue:
                main_queue.append(collector_queue.pop(0))

        num_processed_main = min(len(main_queue), np.random.poisson(mu_main))
        for _ in range(num_processed_main):
            if main_queue:
                main_queue.pop(0)

        main_lengths[t] = len(main_queue)

    return main_lengths

def simulate_collector_server(mu_collector, p, time_steps, collector_queue, processed_memory,
                             edge_queues, k_1):
    collector_lengths = np.zeros(time_steps)

    for t in range(time_steps):
        num_processed_collector = min(len(collector_queue), np.random.poisson(mu_collector))
        for _ in range(num_processed_collector):
            job = collector_queue.pop(0)
            if np.random.rand() < p:
                selected_edge = np.random.randint(1)
                if job in processed_memory[selected_edge]:
                    edge_queues[selected_edge].insert(0, job)
                else:
                    if len(edge_queues[selected_edge]) == k_1:
                        edge_queues[selected_edge].pop()
                    edge_queues[selected_edge].append(job)

            collector_lengths[t] = len(collector_queue)

    return collector_lengths

def simulate_edge_server(mu_edge, lambda_edge, time_steps, edge_queue, processed_memory,
                       collector_queue, discard_rate=0.2, ignore_start_time=0.1):
    edge_lengths = np.zeros(time_steps)

    for t in range(time_steps):
        if t < ignore_start_time * time_steps:
            continue

        num_new_jobs = np.random.poisson(lambda_edge)
        for _ in range(num_new_jobs):
            if np.random.rand() > discard_rate:
                edge_queue.append(f"job_{t}_{_}")

        num_processed = min(len(edge_queue), np.random.poisson(mu_edge))
        for _ in range(num_processed):
            job = edge_queue.pop(0) if edge_queue else None
            if job is not None:
                processed_memory.add(job)
                collector_queue.append(job)

        edge_lengths[t] = len(edge_queue)

    return edge_lengths

def simulate_edge_cloud_system(N=10000, time_end=100, dt=5):
    M = 2
    L_per_collector = [2, 2]
    mu_edge = [[1.0, 1.0], [1.0, 1.0]]
    lambda_edge = [[0.5, 0.5], [0.5, 0.5]]
    mu_collector = [sum(mu) for mu in mu_edge]
    mu_main = sum(mu_collector)
    mu_main *= 0.2
    k_1 = 5

    time_steps = int(time_end / dt)
    edge_lengths = [[0] * time_steps for _ in range(L_per_collector)]
    processed_memory = [[set() for _ in range(L)] for L in L_per_collector]
    collector_queues = [[set() for _ in range(M)] for _ in range(L_per_collector)]
    main_queues = []

    edge_lengths_avg = np.zeros((M, max(L_per_collector), time_steps))
    collector_lengths_avg = np.zeros((M, time_steps))
    main_lengths_avg = np.zeros(time_steps)

    for i in range(N):
        edge_lengths = [np.zeros(L, time_steps) for L in L_per_collector]
        collector_lengths = [np.zeros(M, time_steps)]
        main_lengths = np.zeros(time_steps)

        for t in range(time_steps):
            for j in range(M):
                for l in range(L_per_collector):
                    edge_lengths[l][j][t] = simulate_edge_server(mu_edge[l][j], lambda_edge[l][j], t, dt, lambda_edge[l][j] * dt, time_steps, edge_queues[l][j], processed_memory[l][j], collector_queues[l][j], discard_rate=0.2, ignore_start_time=0.2)

            collector_lengths[j][t] = len(collector_queues[j])

        main_lengths = simulate_main_server(mu_main * dt, time_steps, main_queue, collector_queues)

        edge_lengths_avg += np.array(edge_lengths)
        collector_lengths_avg += collector_lengths
        main_lengths_avg += main_lengths

    edge_lengths_avg /= N
    collector_lengths_avg /= N
    main_lengths_avg /= N

    fig, axes = plt.subplots(1, 1, figsize=(10, 10), sharex=True)
    for j in range(M):
        axes[j].plot(range(time_steps), collector_lengths_avg[j], label=f'Collector {j+1} Queue Length')
        axes[j].set_xlabel('Queue Length')
        axes[j].set_title("Collector Queue Length over Time")
        axes[j].grid()
        axes[j].legend()

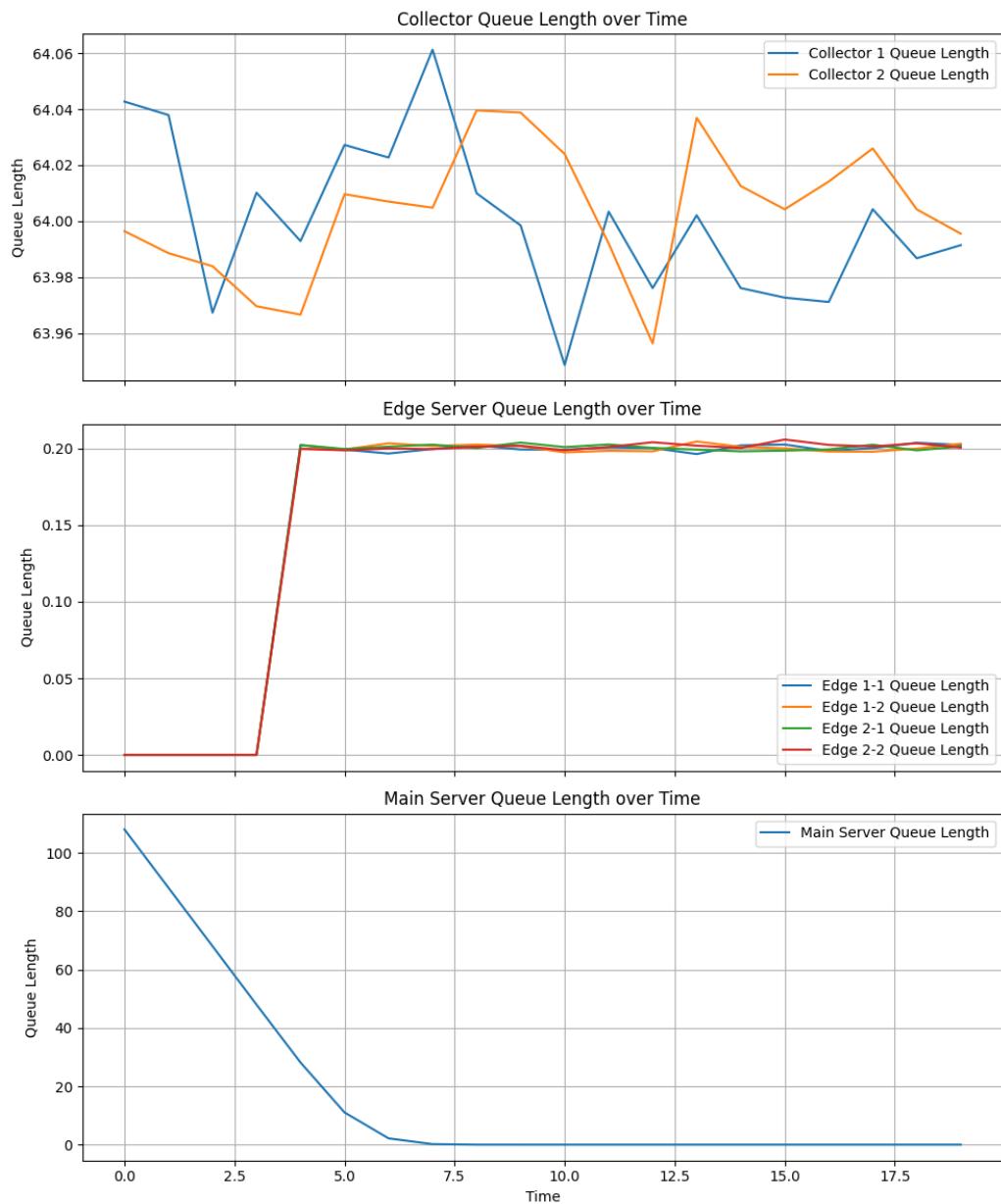
    for l in range(L_per_collector):
        axes[l].plot(range(time_steps), edge_lengths_avg[l], label=f'Edge {l+1}-{k_1} Queue Length')
        axes[l].set_xlabel('Queue Length')
        axes[l].set_title("Edge Server Queue Length over Time")
        axes[l].grid()
        axes[l].legend()

    axes[0].plot(range(time_steps), main_lengths_avg, label="Main Server Queue Length")
    axes[0].set_xlabel('Time')
    axes[0].set_ylabel('Queue Length')
    axes[0].set_title("Main Server Queue Length over Time")
    axes[0].grid()
    axes[0].legend()

    plt.tight_layout()
    plt.show()

simulate_edge_cloud_system()
```

خروجی کد به صورت زیر خواهد بود:



---

## پاسخ پرسش شبیه سازی ۳ :

- نرخ خروجی سرور لبه

$$\lambda_{\text{out,edge}} = \min(\lambda, \mu)$$

$\lambda$  نرخ ورودی هر سرور لبه می باشد.

- نرخ ورودی سرور جمع آوری کننده

$$\lambda_{\text{in,collect}} = 5 \cdot \lambda_{\text{out,edge}}$$

- نرخ ورود موثر سرور جمع آوری کننده

$$\lambda_{\text{eff}} = \lambda_{\text{in,collect}}(1 - p)$$

$\lambda$  نرخ ورود کارها است.  $p$  احتمال بازگشت کارها است.

- نرخ خروجی سرور جمع آوری کننده

$$\lambda_{\text{out,collect}} = \min(\lambda_{\text{eff,collect}}, \mu_{\text{collect}})$$

$$\mu_{\text{collect}} = 5 \cdot \mu_{\text{edge}} = 5 \cdot 10 = 50$$

- نرخ خروجی کل

$$\lambda_{\text{out,system}} = 3 \cdot \lambda_{\text{out,collect}}$$

- مقدار کارهای بازگشته

$$\lambda_{\text{return,total}} = 3 \cdot p \cdot \lambda_{\text{out,collect}}$$

- نرخ از دست دادن کارها

$$\lambda_{\text{lost}} = \max(0, \lambda_{\text{eff}} - L \cdot \mu)$$

حالا نرخ خروج کارها به ازای مقادیر خواسته شده در سوال را محاسبه می کنیم:

$$\mu = 10, \lambda = 3, k = 10, p = 0.1 : 1 \bullet$$

$$\lambda_{\text{out,edge}} = \min(\lambda, \mu) = \min(3, 10) = 3$$

$$\lambda_{\text{in,collect}} = 5 \times \lambda_{\text{out,edge}} = 5 \times 3 = 15$$

$$\lambda_{\text{eff}} = \lambda(1 - p) = 15 \times 0.9 = 13.5$$

$$\lambda_{\text{out,collect}} = \min(\lambda_{\text{eff,collect}}, \mu_{\text{collect}}) = \min(13.5, 50) = 13.5$$

$$\lambda_{\text{out,system}} = 3 \times \lambda_{\text{out,collect}} = 3 \times 13/5 = 40/5$$

$$\mu = 10, \lambda = 6, k = 10, p = 0/1 \quad \lambda_{\text{out,edge}} = \min(\lambda, \mu) = \min(6, 10) = 6 \quad : 2 \bullet$$

$$\lambda_{\text{in,collect}} = 5 \times \lambda_{\text{out,edge}} = 5 \times 6 = 30$$

$$\lambda_{\text{eff}} = \lambda(1 - p) = 30 \times 0/9 = 27$$

$$\lambda_{\text{out,collect}} = \min(\lambda_{\text{eff,collect}}, \mu_{\text{collect}}) = \min(27, 50) = 27$$

$$\lambda_{\text{out,system}} = 3 \times \lambda_{\text{out,collect}} = 3 \times 27 = 81$$

$$\mu = 10, \lambda = 9, k = 10, p = 0/1 \quad : 3 \bullet$$

$$\lambda_{\text{out,edge}} = \min(\lambda, \mu) = \min(9, 10) = 9$$

$$\lambda_{\text{in,collect}} = 5 \times \lambda_{\text{out,edge}} = 5 \times 9 = 45$$

$$\lambda_{\text{eff}} = \lambda(1 - p) = 45 \times 0/9 = 45/5$$

$$\lambda_{\text{out,collect}} = \min(\lambda_{\text{eff,collect}}, \mu_{\text{collect}}) = \min(45/5, 50) = 45/5$$

$$\lambda_{\text{out,system}} = 3 \times \lambda_{\text{out,collect}} = 3 \times 45/5 = 121/5$$

کد این سوال در صفحه بعد آمده است

خروجی کد به صورت زیر است

$$Average \ exit \ rate \ for \ lambda = 3 : 38/67 \bullet$$

$$Average \ exit \ rate \ for \ lambda = 6 : 67/63 \bullet$$

$$Average \ exit \ rate \ for \ lambda = 9 : 96/38 \bullet$$

مشاهده می کنیم به ازای  $\lambda = 3$  نتایج تئوری و شبیه سازی با تقریب تا حد خوبی شباهت دارند ولی به ازای  $\lambda = 6$  و  $\lambda = 9$  اعداد کمی تفاوت دارند ولی باز هم در یک اردر هستند. یعنی با افزایش  $\lambda$  میزان خطای بین نتایج تئوری و شبیه سازی بیشتر می شود.

```

import numpy as np

def simulate_edge_server(mu_edge, lambda_edge, time_steps, edge_queue, processed_memory,
                        collector_queue, discard_rate=0.2, ignore_start_time=0.2):
    edge_lengths = np.zeros(time_steps)
    num_exited = 0

    for t in range(time_steps):
        if t < ignore_start_time * time_steps:
            continue

        num_new_jobs = np.random.poisson(lambda_edge)
        for _ in range(num_new_jobs):
            if np.random.rand() > discard_rate:
                edge_queue.append(f"job_{t}_{_}")

        num_processed = min(len(edge_queue), np.random.poisson(mu_edge))
        for _ in range(num_processed):
            job = edge_queue.pop(0) if edge_queue else None
            if job is not None:
                processed_memory.add(job)
                collector_queue.append(job)
                num_exited += 1

        edge_lengths[t] = len(edge_queue)

    return edge_lengths, num_exited

def simulate_collector_server(mu_collector, p, time_steps, collector_queue, processed_memory,
                             edge_queues, k, L):
    collector_lengths = np.zeros(time_steps)

    for t in range(time_steps):
        num_processed_collector = min(len(collector_queue), np.random.poisson(mu_collector))
        for _ in range(num_processed_collector):
            job = collector_queue.pop(0)
            if np.random.rand() < p:
                selected_edge = np.random.randint(L)
                if job in processed_memory[selected_edge]:
                    edge_queues[selected_edge].insert(0, job)
                else:
                    if len(edge_queues[selected_edge]) > k:
                        edge_queues[selected_edge].pop()
                    edge_queues[selected_edge].append(job)

        collector_lengths[t] = len(collector_queue)

    return collector_lengths

def simulate_main_server(mu_main, time_steps, main_queue, collector_queues):
    main_lengths = np.zeros(time_steps)
    num_exited = 0

    for t in range(time_steps):
        for collector_queue in collector_queues:
            while collector_queue:
                main_queue.append(collector_queue.pop(0))

        num_processed_main = min(len(main_queue), np.random.poisson(mu_main))
        for _ in range(num_processed_main):
            if main_queue:
                main_queue.pop(0)
                num_exited += 1

        main_lengths[t] = len(main_queue)

    return main_lengths, num_exited

def simulate_edge_cloud_system(mu_edge_val, lambda_edge_val, mu_collector_val, mu_main_val, p, k, N,
                              time_end, dt):
    M = 3
    L_per_collector = [5, 5, 5]
    mu_collector = [[mu.edge_val for _ in range(L)] for L in L_per_collector]
    lambda_edge = [[lambda.edge_val for _ in range(L)] for L in L_per_collector]
    mu_collector = [sum(mu) for mu in mu_collector]
    mu_main = mu_main_val

    time_steps = int((time_end / dt))
    edge_queues = [[[] for _ in range(L)] for L in L_per_collector]
    processed_memory = [[{} for _ in range(M)] for L in L_per_collector]
    collector_queues = [[[]] for _ in range(M)]
    main_queue = []
    total_exits = 0

    for _ in range(N):
        edge_lengths = [np.zeros([L, time_steps]) for L in L_per_collector]
        collector_lengths = np.zeros([M, time_steps])
        main_lengths = np.zeros(time_steps)

        total_exited_for_sim = 0

        for t in range(time_steps):
            for j in range(M):
                for l in range(L_per_collector[_]):
                    if len(edge_queues[l][j]) > 0:
                        num_exited_edge = simulate_edge_server(mu_edge_val[l][j], processed_memory[j][l],
                                                               collector_queues[j], discard_rate=0.2, ignore_start_time=0.2)
                        total_exited_for_sim += num_exited_edge
                        collector_lengths[j, t] = len(collector_queues[j])

            main_lengths, num_exited_main = simulate_main_server(mu_main_val * dt, time_steps, main_queue,
                                                               collector_queues)
            total_exited_for_sim += num_exited_main

            total_exits += total_exited_for_sim

        avg_exit_rate = total_exits / (N * time_steps)
        avg_exit_rate = avg_exit_rate / time_end
        return avg_exit_rate

    lambda_values = [3, 6, 9]
    mu.edge_val = 10
    mu.main_val = 10

    p = 0.1
    k = 10
    N = 100
    time_end = 10
    dt = 1

    for lambda_edge_val in lambda_values:
        avg_exit_rate = simulate_edge_cloud_system(mu.edge_val, lambda.edge_val, mu.edge_val, mu.main_val,
                                                p, k, N, time_end, dt)
        print(f"Average exit rate for lambda = {lambda.edge_val}: {avg_exit_rate}")

```

---

پاسخ پرسش شبیه سازی<sup>۴</sup>:

برای بدست آوردن مقدار بهینه  $\mu$  باید تابع هزینه کل را کمینه کنیم.

$$C(\mu) = A \cdot \lambda \cdot (1 - p(\mu)) - A \cdot \mu + B \cdot p(\mu) \cdot \lambda$$

$$p(\mu) = 2 \left( \frac{1}{1 + e^{-0.25\mu}} - 0.5 \right)$$

$$\frac{dC}{d\mu} = - \frac{-0.25e^{-0.25\mu}}{(1 + e^{-0.25\mu})^2} \cdot (2\lambda A + 2\lambda B) - A$$

$$\frac{dC}{d\mu} = 0 \implies A = - \frac{0.25e^{-0.25\mu} \cdot (2\lambda(A+B))}{(1 + e^{-0.25\mu})^2}$$

فرض می کنیم که  $0.8\mu = 0.8$  است.

- اگر  $A > B$  باشد یعنی هزینه از دست دادن یک کار از هزینه برگشت آن به سیستم بیشتر است، یعنی سیستم کاهش از دست دادن کارها را به حداقل کردن بازگشت کارها اولویت خواهد داد. در این حالت باید به دنبال مقدار بیشتر  $\mu$  باشیم چون افزایش  $\mu$  باعث کاهش احتمال از دست دادن کارها می شود. در نتیجه برای به حداقل رساندن کارهای از دست رفته  $\mu$  تا حد امکان بزرگ است.
- اگر  $B < A$  باشد یعنی هزینه بازگشت یک کار از هزینه از دست دادن آن بیشتر است، یعنی سیستم حداقل کردن بازگشت کارها را اولویت قرار خواهد داد. در این حالت سیستم  $\mu$  کوچکتر را ترجیح خواهد داد. در نتیجه  $\mu$  بهینه کوچک است تا احتمال بازگشت کارها کاهش یابد.
- اگر  $A = B$  باشد یعنی هزینه بازگشت کارها و تعداد کارهای از دست رفته برابر است و بهینه سازی باید این دو عامل را متعادل کند. برای این حالت باید یک  $\mu$  انتخاب شود که بازگشت و از دست دادن کارها را متعادل کند کد این بخش در صفحه بعد آمده است.

خروچی این کد به صورت زیر است:

$$Optimal \quad Rate \quad for \quad A = 1, B = 0.5 : 1/39 \quad •$$

$$Optimal \quad Rate \quad for \quad A = 1, B = 1 : 2/16 \quad •$$

$$Optimal \quad Rate \quad for \quad A = 0.5, B = 1 : 1/00 \quad •$$

مشاهده می کنیم نتایج حاصل از شبیه سازی با تحلیل های تئوری تاحد خوبی مطابقت دارد.

```

import numpy as np
import matplotlib.pyplot as plt

class TaskProcessor:
    def __init__(self, rate, max_queue_size, num_primary_workers, num_secondary_workers,
                 retry_probability):
        self.rate = rate
        self.max_queue_size = max_queue_size
        self.num_primary_workers = num_primary_workers
        self.num_secondary_workers = num_secondary_workers
        self.retry_probability = retry_probability
        self.worker_queues = [[[] for _ in range(num_secondary_workers)] for _ in
                             range(num_primary_workers)]
        self.primary_queues = [[] for _ in range(num_primary_workers)]
        self.final_task_queue = []
        self.next_task_completion = [[np.inf for _ in range(num_secondary_workers)] for _ in
                                     range(num_primary_workers)]
        self.next_primary_completion = [np.inf for _ in range(num_primary_workers)]
        self.next_final_task_completion = np.inf
        self.lost_task_count = 0
        self.retried_task_count = 0
        self.completed_task_count = 0

    def transition_probability(self):
        return 2 * (1 / (1 + np.exp(-0.25 * self.rate)) - 0.5)

    def handle_new_task(self, task_arrival):
        primary_worker_idx = np.random.randint(self.num_primary_workers)
        secondary_worker_idx = np.random.randint(self.num_secondary_workers)

        if len(self.worker_queues[primary_worker_idx][secondary_worker_idx]) < self.max_queue_size:
            self.worker_queues[primary_worker_idx][secondary_worker_idx].append(task_arrival)
        else:
            self.lost_task_count += 1

        if len(self.worker_queues[primary_worker_idx][secondary_worker_idx]) == 1:
            self.next_task_completion[primary_worker_idx][secondary_worker_idx] =
            np.random.exponential(1 / self.rate)

    def process_secondary_worker_completion(self, current_time):
        for primary_idx in range(self.num_primary_workers):
            for secondary_idx in range(self.num_secondary_workers):
                if current_time == self.next_task_completion[primary_idx][secondary_idx]:
                    self.worker_queues[primary_idx][secondary_idx].pop(0)
                    if np.random.rand() < self.retry_probability:
                        self.retried_task_count += 1
                        if len(self.worker_queues[primary_idx][secondary_idx]) < self.max_queue_size:
                            self.worker_queues[primary_idx][secondary_idx].append(current_time)
                        else:
                            self.lost_task_count += 1
                    else:
                        self.primary_queues[primary_idx].append(current_time)

                    if self.worker_queues[primary_idx][secondary_idx]:
                        self.next_task_completion[primary_idx][secondary_idx] = np.random.exponential(1 /
                        self.rate)
                    else:
                        self.next_task_completion[primary_idx][secondary_idx] = np.inf

                    if len(self.primary_queues[primary_idx]) == 1:
                        self.next_primary_completion[primary_idx] = np.random.exponential(1 /
                        self.rate)

    def process_primary_worker_completion(self, current_time):
        primary_idx = self.next_primary_completion.index(current_time)
        self.primary_queues[primary_idx].pop(0)
        self.final_task_queue.append(current_time)

        if self.primary_queues[primary_idx]:
            self.next_primary_completion[primary_idx] = np.random.exponential(1 / self.rate)
        else:
            self.next_primary_completion[primary_idx] = np.inf

        if len(self.final_task_queue) == 1:
            self.next_final_task_completion = np.random.exponential(1 / self.rate)

    def process_final_task_completion(self):
        self.final_task_queue.pop(0)
        self.completed_task_count += 1

        if self.final_task_queue:
            self.next_final_task_completion = np.random.exponential(1 / self.rate)
        else:
            self.next_final_task_completion = np.inf

    for (param_A, param_B), optimal_rate in optimal_mus.items():
        print(f'Optimal Rate for A={param_A}, B={param_B}: {optimal_rate:.2f}')

```