COMP 5320/6320/6320-D1 - Design and Analysis of Computer Networks
Course Project Report: Two-Queue System Performance Analysis

## Introduction

In this project, we simulated a two-queue system to compare different packet assignment strategies. The system has two queues, each with a capacity of 10 packets (including the one being served). We tested two strategies: random selection and min-queue selection. Both queues use FCFS service.

We evaluated three performance metrics:
1. Blocking Probability - ratio of dropped packets to total packets offered
2. Average Queue Length - mean number of packets in queue, sampled at arrivals
3. Average Sojourn Time - mean time packets spend in the system (only admitted packets)

We varied three parameters:
- Arrival Rate ($\lambda$) - Poisson arrivals
- Service Rate ($\mu$) - exponential service times
- Traffic Load ($\rho = \lambda/(2\mu)$)

## Implementation

We used an event-driven simulation approach. Instead of updating continuously, the simulator jumps between important events (packet arrivals and departures). This is more efficient than continuous time simulation.

The two event types are:
- ARRIVAL - a new packet enters the system
- DEPARTURE - a packet finishes service and leaves

I used a priority queue (Python's heapq) to manage events ordered by time. Each queue tracks the packets waiting and the one currently being served.

The two packet assignment strategies work like this:

Random Strategy: Each packet is assigned to queue 1 or queue 2 with 50% probability.
This creates two independent M/M/1/K queues with arrival rate $\lambda/2$ each, which lets
us validate against theoretical results.

Min-Queue Strategy: Each packet goes to whichever queue is shorter at the moment it arrives. This should balance the load better but the queues are no longer independent, so there's no closed-form solution to compare against.

Each simulation ran 10,000 packets, and I averaged results across 10 different random seeds to get smooth curves and reduce noise.

## Theoretical Results for Random Strategy

For the random selection strategy, each queue behaves as an independent M/M/1/K queue with K=10 and arrival rate $\lambda/2$. Let $\rho_q = (\lambda/2)/\mu$.

The formulas are:

Blocking Probability:
$P_{block} = (1 - \rho_q) \times \rho_q^K / (1 - \rho_q^{(K+1)})$

Average Queue Length:
$E[N] = \rho_q \times (1 - (K+1)\rho_q^K + K\rho_q^{(K+1)}) / ((1 - \rho_q)(1 - \rho_q^{(K+1)}))$

Average Sojourn Time (using Little's Law):
$E[T] = E[N] / (\lambda_{eff})$ where $\lambda_{eff} = (\lambda/2) \times (1 - P_{block})$

When $\rho_q = 1$:
$P_{block} = K/(K+1) = 10/11$
$E[N] = K/2 = 5$

The min-queue strategy doesn't have theoretical results because the queues are correlated (not independent anymore).

## Our Results

### Effect of Arrival Rate

Figures 1-3 show how performance changes with arrival rate when $\mu=1.0$.

Figure 1 (Blocking Probability):
At low $\lambda$ (below 1.0), both strategies have almost no blocking. At medium $\lambda$ (1.5-2.5), min-queue shows 50-70% lower blocking than random. At high $\lambda$ (above 3.0),
both strategies start to saturate and blocking gets high. The simulated random strategy matches the theoretical curve within about 2% error, which confirms the implementation is correct.

Figure 2 (Average Queue Length):
At low load, both strategies keep queues short (1-2 packets). Interestingly, min-queue actually has slightly longer queues at medium load. This happens because
min-queue admits more packets overall (less blocking), so more packets are in the
system. At high load, random saturates faster.

Figure 3 (Average Sojourn Time):
Sojourn time grows exponentially with arrival rate for both strategies. Min-queue
has 10-20% lower sojourn time at medium loads. Both strategies show a dramatic increase when $\lambda > 2.5\mu$.

### Effect of Service Rate

Figures 4-6 show performance vs service rate with fixed $\lambda=2.0$.

Figure 4 (Blocking Probability):
At low $\mu$ (below 1.0), the system is overloaded and blocking exceeds 40%. There's a sharp decline in blocking as $\mu$ increases from 1.2 to 1.8. At high $\mu$ (above 2.5),
both strategies achieve less than 5% blocking. Min-queue provides about 60% improvement in the moderate service rate range.

Figure 5 (Average Queue Length):
Queue length decreases as service rate increases, following a hyperbolic curve. Min-queue maintains higher queue lengths at all service rates, again because it accepts more packets.

Figure 6 (Average Sojourn Time):
Sojourn time has an inverse relationship with service rate. Min-queue shows a consistent 15% improvement over random. Both strategies converge to around $1/\mu$ (pure service time) at high $\mu$.

### Effect of Traffic Load

Figures 7-9 show performance vs traffic load $\rho = \lambda/(2\mu)$ with $\mu$=1.0.

Figure 7 (Blocking Probability):
There's a critical threshold around $\rho = 0.85$ where blocking starts to rise sharply.
Min-queue delays this threshold to around $\rho = 1.1$. Above $\rho = 1.5$, both strategies
experience over 35% blocking. The biggest gap between strategies is at $\rho = 1.0$ to 1.3.

Figure 8 (Average Queue Length):
Linear growth at low loads ($\rho < 0.7$), exponential growth at medium loads
($0.7 < \rho < 1.2$), and saturation toward capacity at high loads ($\rho > 1.5$).
Min-queue consistently shows 10-15% higher queue occupancy.

Figure 9 (Average Sojourn Time):
Sojourn time stays low (under 2 time units) for $\rho < 0.8$, then increases rapidly
in the range $0.8 < \rho < 1.4$. Both strategies show unbounded growth as $\rho$ approaches 2.
Min-queue provides about 20% improvement at $\rho = 1.0$.

Traffic load $\rho$ is probably the most useful parameter since it directly represents
how loaded the system is. The critical operating region is $0.8 < \rho < 1.3$ where
queueing effects are strongest but the system hasn't completely saturated.

## Sensitivity Analysis

To figure out which parameters matter most, I looked at how much each metric
changes when you vary each parameter.

The ranking from highest to lowest impact is:

1. Traffic Load ($\rho$) - Highest Impact
Traffic load has by far the biggest impact on all metrics. A 10% increase in $\rho$
can cause a 40-50% increase in blocking probability and sojourn time when you're
near $\rho = 1.0$. This makes sense because $\rho$ captures the fundamental balance between
arrivals and service.

2. Arrival Rate ($\lambda$) - High Impact
Arrival rate has strong impact since it directly controls the offered load. The
effect is nearly linear at low loads but exponential at high loads.

3. Service Rate ($\mu$) - Moderate Impact
Service rate has an inverse relationship with the metrics. Doubling the service
rate approximately halves sojourn time and blocking probability.

Key findings:
- The system has a phase transition around $\rho = 0.8$-1.0 where behavior changes from "lightly loaded" to "heavily loaded"
- Min-queue strategy helps most in the medium load region ($0.8 < \rho < 1.3$)
- At very low loads both strategies are basically the same, and at very high loads both saturate
- Blocking probability is more sensitive than queue length, so it's the most important metric for capacity planning

## Conclusions

The main takeaways from this project:

1. Min-queue strategy consistently beats random selection. At moderate traffic

loads (ρ = 0.8-1.3), it reduces blocking by 50-70% and sojourn time by 15-20%.

2. Traffic load ρ is the most important parameter. The critical region is
0.8 < ρ < 1.3 where queueing dynamics are most pronounced.

3. The simulation matches theory for the random strategy (within 2% error),
which validates that the implementation is correct.

4. Min-queue works by dynamically balancing load between queues. The tradeoff is
that it actually has slightly higher queue occupancy (because it admits more
packets
instead of dropping them).

For practical networks:
- Keep ρ below 0.7 for stable, low-latency operation
- Use load balancing (like min-queue) when 0.7 < ρ < 1.2
- Add capacity if ρ consistently exceeds 1.0

The finite capacity (K=10) causes blocking to become significant when ρ > 0.8.
With infinite capacity there'd be no blocking but sojourn times would grow
unbounded.

Implementation Notes

Language: Python 3
Libraries: heapq (priority queue), numpy (statistics), matplotlib (plots)

Key design choices:
- Event times generated using exponential distribution (inter-arrival =
-ln(U)/λ,
service = -ln(U)/μ)
- Queue length sampled at each arrival event for both queues
- Sojourn time tracked per-packet as departure_time - arrival_time
- Used seeds 0-9 for the 10 independent runs

Each simulation run processes about 20,000 events (10k packets + departures),
which takes about 0.1 seconds. Total runtime for all experiments is around 90
seconds.