Event Stack Software Project

CSE 517: Performance Evaluation

Arryan Bhatnagar
991790192
apb6254@psu.edu
November 8, 2024

Question 1

How to generalize your simulation to more arbitrary queueing networks. In particular, how to simulate more general renewal arrivals (easy) and how to simulate a Jackson network or a network with fixed paths between end-points, including a nice way for the user to specify: the network topology and paths, the inter-arrival time and service time distributions as chosen from the broader family of exponential phase-type, and the performance measures.

In the initial setup, my simulation is designed for a *tandem queue* model, where jobs flow sequentially from one queue to the next. This model is represented as a simple array of Queue objects, with each queue connected linearly to the next:

```
queues = [
    Queue(queue_id=1, service_rate=0.5),
    Queue(queue_id=2, service_rate=0.7),
    Queue(queue_id=3, service_rate=0.6),
    Queue(queue_id=4, service_rate=0.4),
    Queue(queue_id=5, service_rate=0.1),
]
```

To extend this setup into a more *arbitrary queueing network,* one where jobs can move between multiple queues in different patterns, it is essential to introduce a new structure that supports more complex topologies, flexible routing mechanisms, and different distributions.

1. Network Data Structure:

In a tandem queue setup, queues are defined in a one-dimensional array, which limits each job to follow a single, linear path. To allow more complex structures, I would implement a dictionary-based network data structure, where each queue can route jobs to multiple other queues with specific probabilities. This dictionary would hold the configuration of each queue, including its service rate and routing probabilities for each potential destination queue.

A sample configuration might look like this:

```
network = {
    1: {'service_rate': 0.5, 'routes': {2: 0.7, 3: 0.3}},
    2: {'service_rate': 0.7, 'routes': {4: 1.0}},
    3: {'service_rate': 0.6, 'routes': {4: 1.0}},
    4: {'service_rate': 0.4, 'routes': {}},
} # dynamic routing not shown

# Queue 1 routes to Queue 2 (70%) and Queue 3 (30%)
# Queue 2 routes to Queue 4 (100%)
# Queue 3 routes to Queue 4 (100%)
# Queue 4 is an endpoint with no further routing
```

- Each queue is assigned a service rate that determines how quickly it processes jobs.
- The `routes` dictionary specifies possible next queues and the probability of selecting each route. This design supports both probabilistic routing and fixed paths by defining specific probabilities for each route.

2. Routing Logic:

To incorporate flexible routing, there are multiple different mechanisms to achieve this such as:

- **Fixed Routing:** Each job has a predetermined path through the network (e.g., a job exiting queue 1 will always proceed to queue 7)
- **Probabilistic Routing:** Probabilities are used to determine the next queue, allowing jobs to move to different queues based on defined likelihoods. This routing can be implemented with Python's `random.choices` method to select the next queue based on the specified probabilities and will be shown later. (e.g., a job exiting queue 1 has a 30% chance to proceed to queue 2 and a 70% chance to proceed to queue 3)
- **Dynamic Routing:** In a dynamic routing setup, routing decisions are made based on the current state of the queues, such as choosing the queue with

the shortest current length. While dynamic routing is not shown in the example, it could be incorporated by assessing the state of each possible destination queue before selecting a path.

The routing function below demonstrates probabilistic routing using the `network` dictionary:

```
    # Get the routes and their probabilities
    routes = network[current_queue_id][‘routes’]

    if routes:
        next_queue_id = random.choices(
            list(routes.keys()),    # List of queues
            weights=routes.values() # Corresponding probabilities
       )[0] # Extract the selected queue ID from the list
        return next_queue_id
    else:
        return None  # No further routes, end of path
```

3. Service and Inter-Arrival Time Distributions

Service Time Distributions:
In Jackson networks, service times are typically exponential to maintain the memoryless property, but this can be generalized. Supporting other phase-type distributions, such as Erlang or hyper-exponential, would allow for scenarios where service times vary while still being part of the exponential phase-type family. The below change of the `generate_service_time` function shows support for Erlang distributions

Before:

```
def generate_service_time(self)
    return random.expovariate(self.service_rate)
```

After:

```
def generate_service_time(queue_id):
    service_rate = network[queue_id]['service_rate']
    return random.expovariate(service_rate)

def generate_service_time(queue_id, distribution='exponential',
       params=None):

    service_rate = network[queue_id]['service_rate']

    if distribution == 'exponential':
        return random.expovariate(service_rate)
    elif distribution == 'erlang':
        k = params['k'] # Erlang shape parameter
        service_time = 0
        for _ in range(k):
            service_time += random.expovariate(service_rate)
        return service_time
```

This function accommodates multiple service time distributions, including exponential and Erlang, providing flexibility in representing different job service characteristics.

Inter-Arrival Time Distributions:
Similarly, the expanded support of inter-arrival will follow a similar code in the `generate_poisson_arrival` function which would also have it's name changed to something like `generate_arrival_time` but will follow the same structure as the `generate_service_time` function.

4. Performance Measures

To accurately assess performance in an arbitrary queueing network, we need to expand the range of performance metrics. There are some metrics which might be useful in both, tandem queues and arbitrary queueing networks, and some that will be useful only in arbitrary queueing networks.

Metric	Both	Mostly for Arbitrary Queue Networks
Total System Throughput	✓	
Variance of Queue Length	✓	
Job Drop Rate	✓	
Queue Jump Frequency		✓
Cumulative Throughput		✓
Path Sojourn Time		✓
Load Distribution across Paths		✓

Metrics Useful in Both Tandem and Arbitrary Queueing Networks

- **Total System Throughput**: The rate at which jobs complete the system. Throughput is critical in understanding the network's capacity and is useful in both sequential and branching networks.
- **Variance of Queue Length and Sojourn Time:** The variability in queue length and sojourn time across the network. This measure provides insights into the stability of the system, indicating if certain queues have highly variable wait times or load.
- **Job Drop Rate or Blocking Probability (if applicable):** The proportion of jobs that fail to enter a queue due to capacity limits or are dropped due to system constraints. While less common in tandem queues, this can be essential in more complex networks.

Metrics Primarily Useful in Arbitrary Queueing Networks

In arbitrary queueing networks, where jobs may follow branching, probabilistic, or dynamic paths, additional metrics help capture the complexity and routing behavior:

- **Queue Jump Frequency (Routing Pattern Analysis):** This tracks the frequency of job transitions between different queues (e.g., Queue 1 → Queue 4 or Queue 2 → Queue 5). While not a traditional metric, tracking queue jumps provides valuable insights into routing patterns and is particularly helpful in complex networks. This metric can highlight commonly used paths, help identify routing bottlenecks, and assist in load balancing by indicating which routes are overused.
- **Cumulative Throughput per Queue:** In an arbitrary network, cumulative throughput per queue helps identify bottlenecks and heavily used queues. By tracking cumulative throughput at each queue, we gain insights into which parts of the network are more congested or heavily trafficked.
- **Path Sojourn Time:** The average time jobs spend on specific paths through the network, from entry to exit. This metric helps identify paths with longer delays and assists in optimizing the routing configuration for improved performance.
- **Load Distribution across Paths:** This measures the distribution of jobs across different paths in the network, offering insights into load balancing effectiveness and path utilization.

Question 2

When the performance measures are mean network sojourn times, explain a role (if any) for Little's formula.

Little's formula says that in a stationary, lossless, and stable queuing system:

$$L = \lambda \cdot w$$

Where,
L: Average number of jobs in the system (or queue length).
λ: Average arrival rate of jobs into the system.
w: Average time a job spends in the system, also known as mean sojourn time.

In a tandem queue configuration, where jobs progress sequentially through each queue in the network, Little's Law applies both at the individual queue level and across the entire network.

Role of Little's Formula at the Queue Level
For each queue i Little's Law holds:

$$L_i = \lambda_i \cdot w_i$$

Role of Little's Formula at the Network Level

$$L = \lambda \cdot w$$

Question 3

Finally, one may partition the network and a have different event stack for each partition element. In this case, a stack X may receive an event from another stack that precedes events in stack X. Describe what needs to be done (if anything) and how that would change the operation of the stack - hint: rollback.

We need to perform timestamp validation. When an event from another stack arrives in stack X, check its timestamp against the timestamps of the events already in stack X. If the new event's timestamp is earlier than any event in stack X, a rollback may be necessary to reestablish the correct chronological order. This would mean that we insert the incoming event into stack X at the appropriate position based on its timestamp, moving any later events forward in the stack. This ensures that events are processed in the correct temporal sequence.

For sure the rollback mechanism introduces complexity, as each stack must be able to dynamically adjust to incoming events from other stacks. It will mainly increase processing time due to the need for frequent timestamp checks and potential reordering.

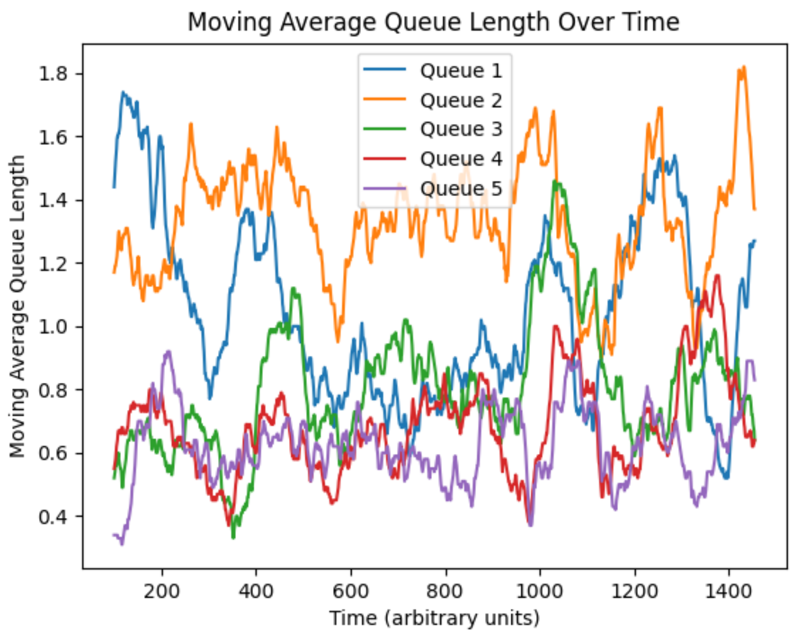
Simulations

The queue configuration used is:

```
queues = [
    Queue(queue_id=1, service_rate=3.5),
    Queue(queue_id=2, service_rate=2.7),
    Queue(queue_id=3, service_rate=5.6),
    Queue(queue_id=4, service_rate=6.4),
    Queue(queue_id=5, service_rate=7.1),
]
arrival_rate = 3.0
```

Moving Average Queue Length Over Time

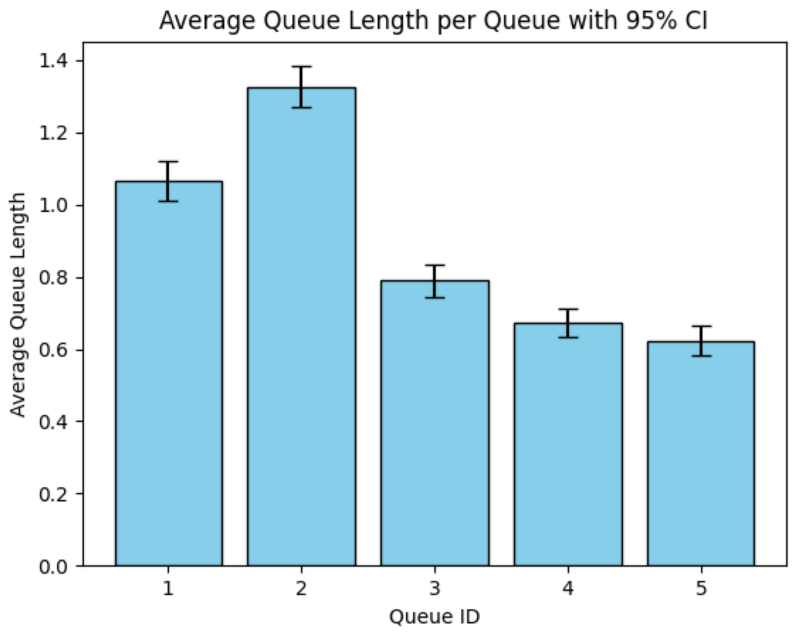
The plot captures the fluctuations in queue lengths over time. Queue lengths fluctuate around their average values, reflecting the randomness in arrivals and service completions.



Average Queue Length per Queue with 95% Confidence Interval

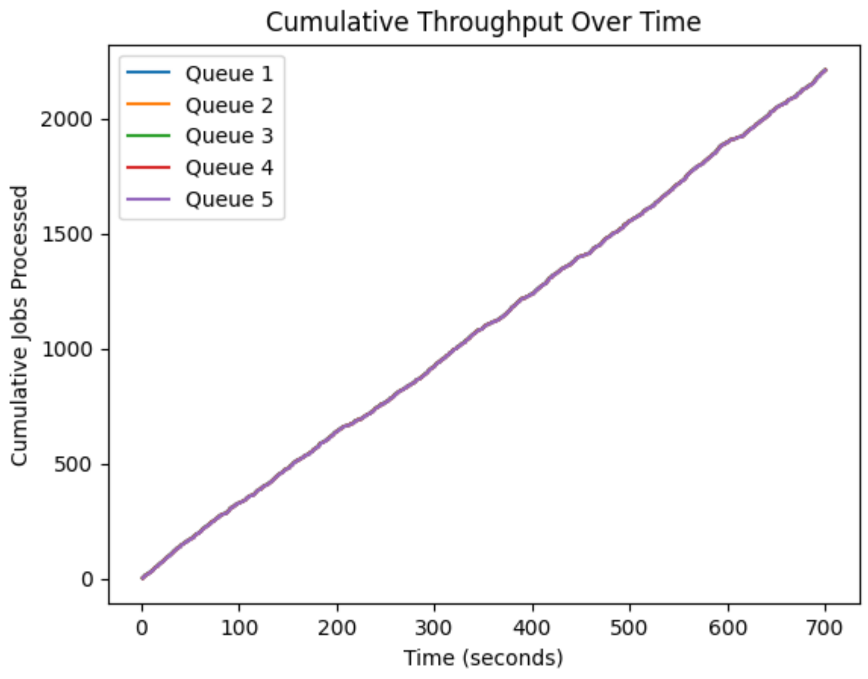
The Average Queue Length per Queue plot, with error bars showing the 95% confidence interval, highlights the stability of queue lengths:

- Queue 1: 1.06 ± 0.06
- Queue 2: 1.33 ± 0.06
- Queue 3: 0.79 ± 0.05
- Queue 4: 0.67 ± 0.04
- Queue 5: 0.62 ± 0.04



Cumulative Throughput Over Time

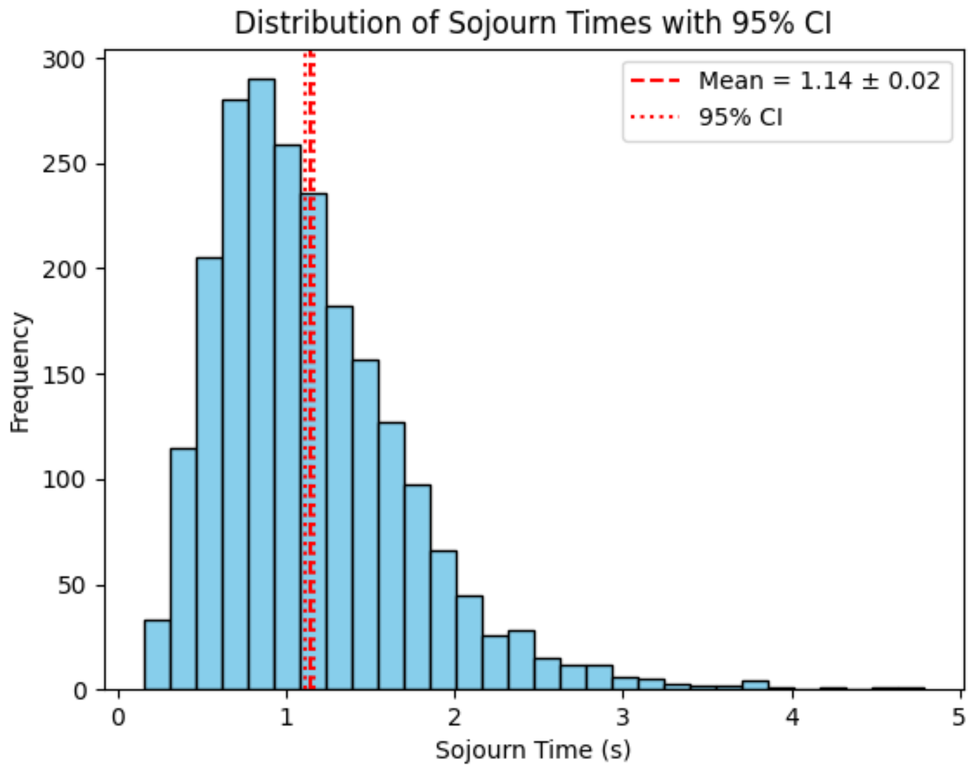
The Cumulative Throughput plot demonstrates the steady increase in job completions across all queues. Each queue processed over 2200 jobs, confirming high throughput and effective utilization.



Distribution of Sojourn Times with 95% Confidence Interval

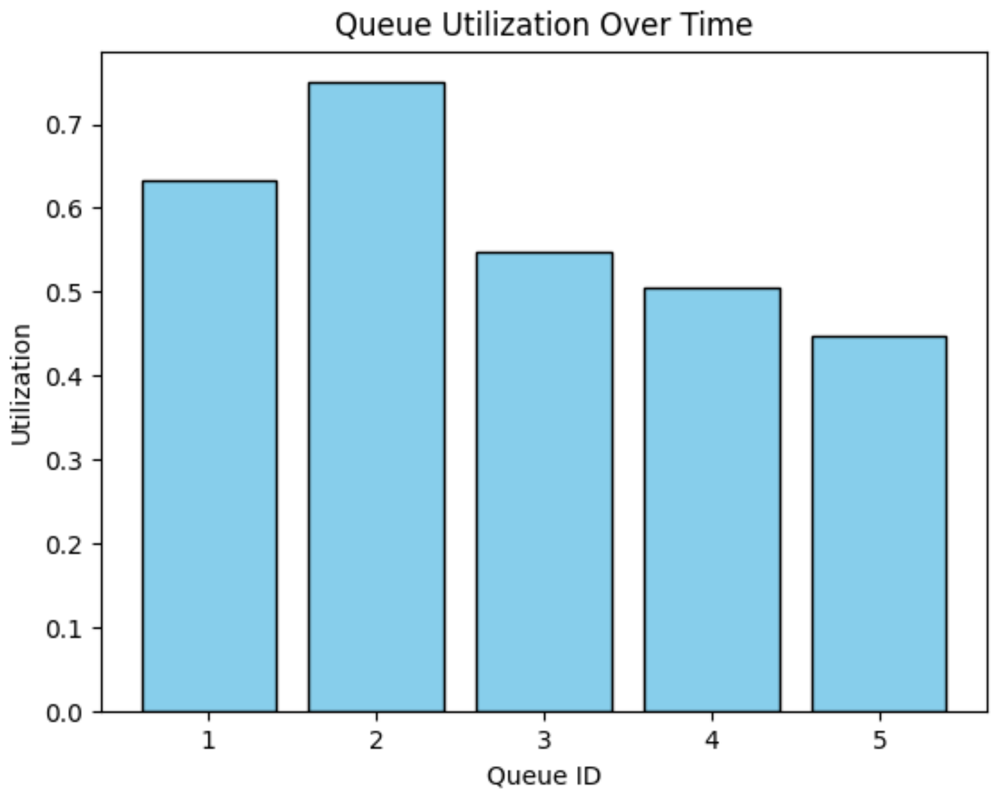
The Sojourn Time Distribution plot shows the time jobs spend in the system from arrival to completion. The histogram displays a right-skewed distribution, which is expected due to the variability in processing times.

- Mean Sojourn Time: 1.14 ± 0.02 seconds, close to the theoretical value of 1.13 seconds as predicted by Jackson's Theorem.



Queue Utilization Over Time

The Queue Utilization plot shows the proportion of time each queue is busy. The utilization values match expected load levels, with Queue 2 showing the highest utilization at around 0.74, due to its relatively low service rate.



For an M/M/1 queue, utilization ρ is calculated as:

$$\rho = \frac{\lambda}{\mu}$$

- Queue 1: ≈ 0.857
- Queue 2: ≈ 1.111
- Queue 3: ≈0.536
- Queue 4: ≈0.469
- Queue 5: ≈0.423