CSE 517 Fall 2024 Simulation Project – Tandem Queues

Student Name : Vipul Sachdeva

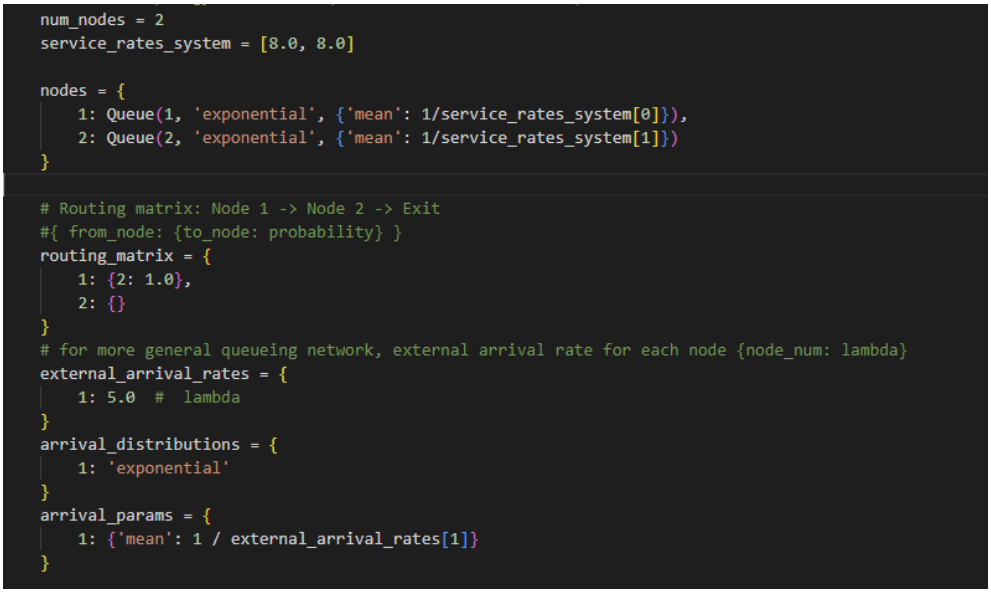
PSU ID: 949610346

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.**

   To simulate arbitrary queueing networks, the network system is structured using routing matrices that define the network topology and the paths between nodes using matrix like data structure matrices for specifying easy configuration.

   - For my simulation, I have used a simple python dictionary to specify the nodes in the system where key number is the queue(node) number and value is Queue object with the distribution type of service times and required parameters.
   - Users can specify fixed paths by setting the routing probability from one node to the next to one, ensuring that jobs follow a predetermined sequence through the network. To simulate renewal processes, routing matrices can be altered to route a job back to a previous node after its completion with a specified probability. Again, I have used dictionary of dictionary to specify the routing matrix
   - To accommodate more general arrival intervals and service times, the generate_random_sample(...) function can be modified to add more distribution types and allow users to select from family of distributions, such as exponential, poison, erlang, coxian, lognormal, beta etc. by simply specifying the desired distribution type and parameters. This function is used inside the simulation loop to generate the arrival and departure time of the event in the event stack. This enables the simulation of a variety of arrival processes and service mechanisms.
   - Upon completion of the simulation, a comprehensive list recorded metrics is returned, like mean sojourn time, queue lengths, and utilizations. These metrics can be accessed and visualized through built-in plotting functions, providing users with insightful graphs and statistical summaries.

Eg of network topology in my simulation



```
num_nodes = 2
service_rates_system = [8.0, 8.0]

nodes = {
    1: Queue(1, 'exponential', {'mean': 1/service_rates_system[0]}),
    2: Queue(2, 'exponential', {'mean': 1/service_rates_system[1]})
}

# Routing matrix: Node 1 -> Node 2 -> Exit
#{ from_node: {to_node: probability} }
routing_matrix = {
    1: {2: 1.0},
    2: {}
}
# for more general queueing network, external arrival rate for each node {node_num: lambda}
external_arrival_rates = {
    1: 5.0  #  lambda
}
arrival_distributions = {
    1: 'exponential'
}
arrival_params = {
    1: {'mean': 1 / external_arrival_rates[1]}
}
```

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

Little's formula relates the average number of jobs in a system (L) to the average arrival rate (λ) and the average time a job spends in the system: $L=\lambda W$.

In the context of mean network sojourn times, little's formula is used cross-verify simulation results. By ensuring that the simulated mean sojourn times satisfy Little's relationship, the consistency and accuracy of the simulation can be validated.

For my current simulation:

- **L**: Total average number of jobs in the system (across all queues).
- **λ**: Arrival rate of jobs to the system = 5 jobs/second
- **W**: Mean sojourn time (simulated) = 0.6811 seconds ± 0.0055 [0.99 Confidence level]
- **Applying Little's Law**:

  L= λ*W = 5×0.6811= **3.4057 ± 0.027 jobs**

The simulation result for average jobs in the system is (3.4309 ± 0.0249) which matches closely with the theoretical calculation, validating the simulation's accuracy.

3.One may partition the network and 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.

While specifying the nodes in the network we can create specified partitions, and each partition can have eventstack with stack_id and any number of nodes within itself. Routing probabilities can be same
Eg.

```
nodes = {
    1: Queue(1, 'exponential', {'mean': 1/service_rates_system[0]}),
    2: Queue(2, 'exponential', {'mean': 1/service_rates_system[1]})
}
#can be changed to
partitioned_nodes = {
    1: {
        'event_stack': EventStack(1),
        'nodes': {
            1: Queue(1, 'exponential', {'mean': 1/service_rates_system[0]}),
        }
    },
    2: {
        'event_stack': EventStack(2),
        'nodes': {
            2: Queue(2, 'exponential', {'mean': 1/service_rates_system[1]}),
        }
    }
}
```

When partitioning a network simulation with separate event stacks for each partition, if an event arrives at Stack X from another stack with a timestamp earlier than events already scheduled in Stack X, Stack X must perform a rollback to maintain causality. This will be reverting the simulation state within Stack X to a point just before the timestamp of the incoming event. It requires restoring the previous state of the system and discarding any events

that were scheduled after the rollback time. To have this additional failsafe mechanism, the stack must maintain snapshots of its state (such as node statuses being busy or idle, queue lengths) at various points in time, allowing it to restore the system. Once rolled back, Stack X discards any events that were scheduled after the rollback time to prevent inconsistencies. After restoration, the incoming event would be inserted into the correct chronological position, and any subsequent events are re-evaluated in the proper chronological order, and then continue processing future events. This changes the operation of the event stack by introducing state management capabilities, where the stack must save snapshots of the system state at various times and handle the reprocessing of events after a rollback. Implementing rollbacks is essential to maintain consistency and accuracy across different partitions.

The main changes in code would be to have a source stack id property of an Event , an additional stack_id, and stack_history for eventStack to save checkpoints.

Simulation Results:

1. Poisson Arrivals and Service Time Distributions

The main simulation run 10000 sim seconds with external arrival rate at 5 jobs/second, and varying service time from 4 to 10 jobs/ second. The remaining graphs (node length, batch means, histogram and theoretical vs simulated mean comparison) are for constant service time of 8 jobs/second with all metrics and plots showing a steady state (not much variation in queue lengths and sojourn times). It can be seen that when μ ≤ 5 (arrival rate), the queue lengths grow indefinitely, and the system is unstable as there is no convergences towards mean.

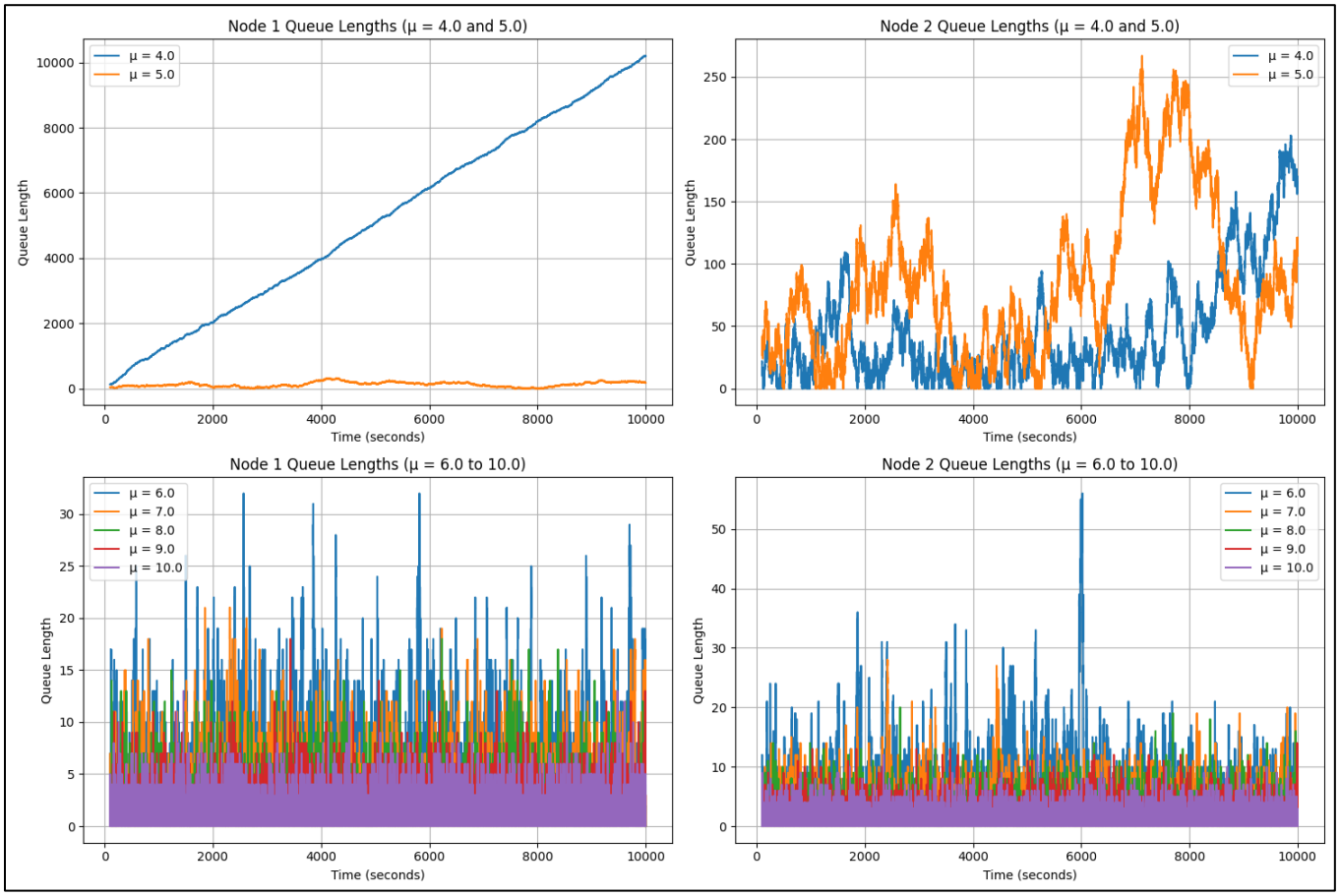


Figure 1: Wait Queue Lengths at Node 1 and 2 with different service times

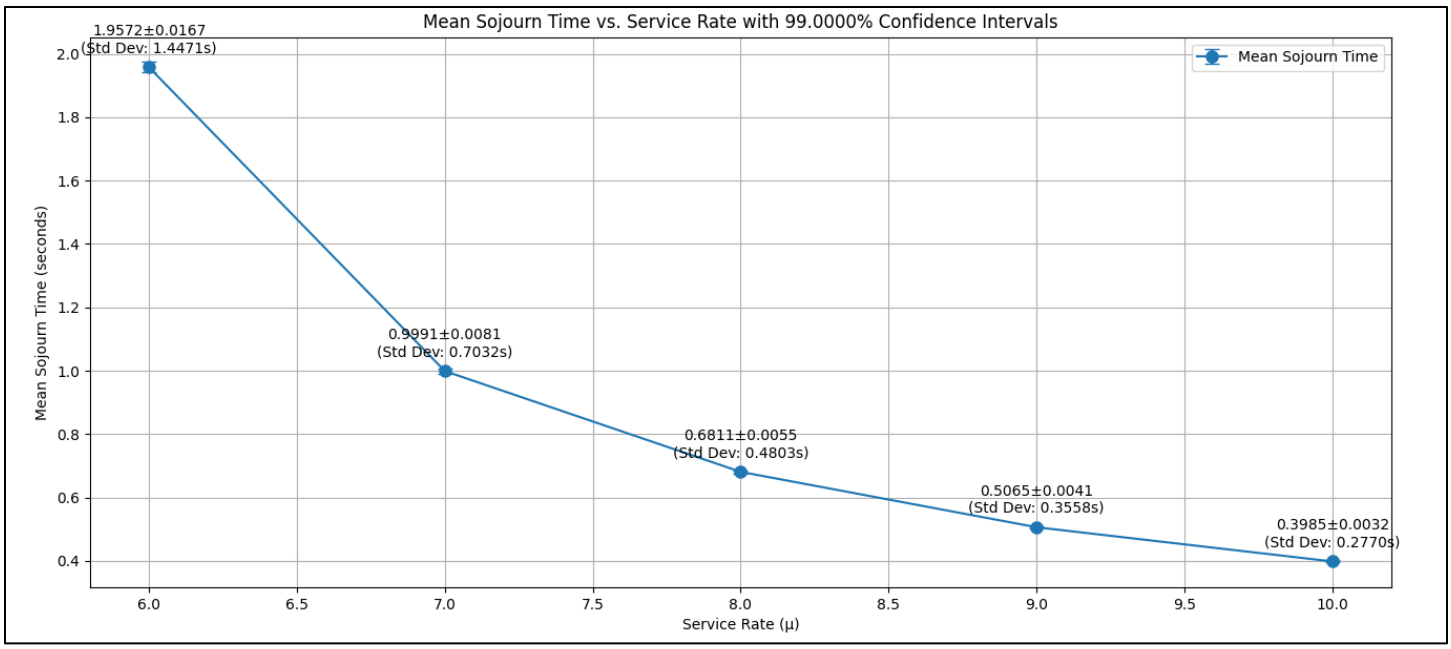


Figure 2: Sojourn times vs Nodes Service Rate with fixed arrival rate (5.0)

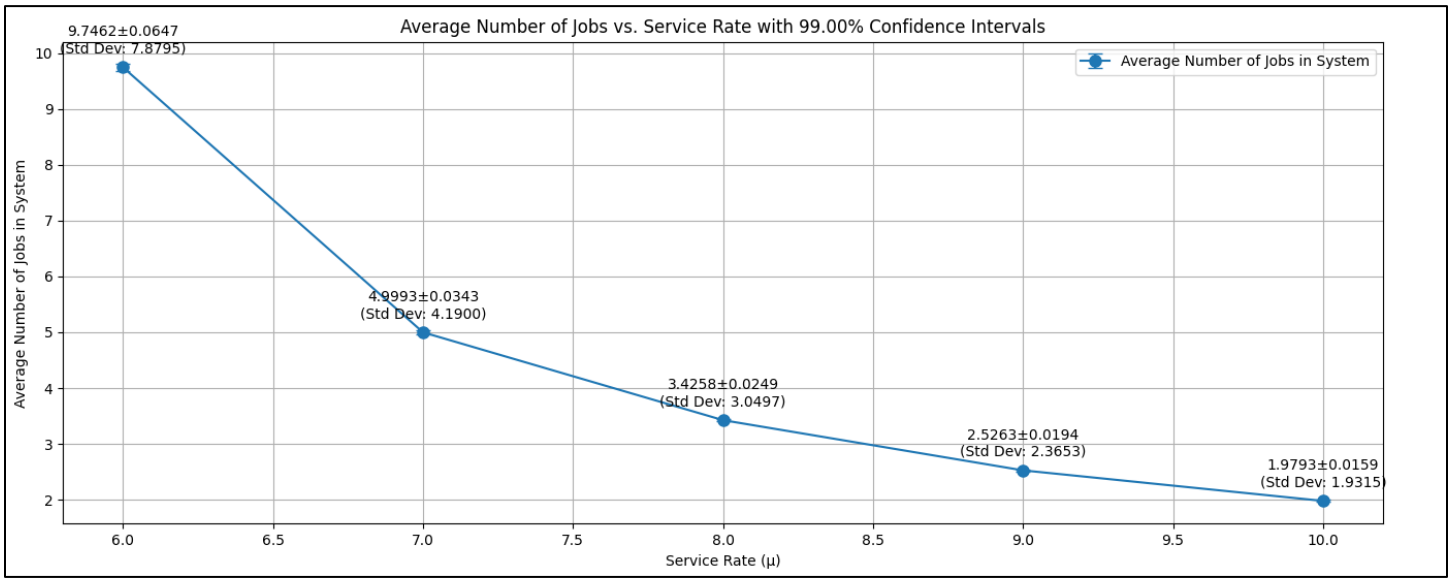


Figure 3:Average no of jobs vs Nodes service rates with fixed arrival rate (5.0)

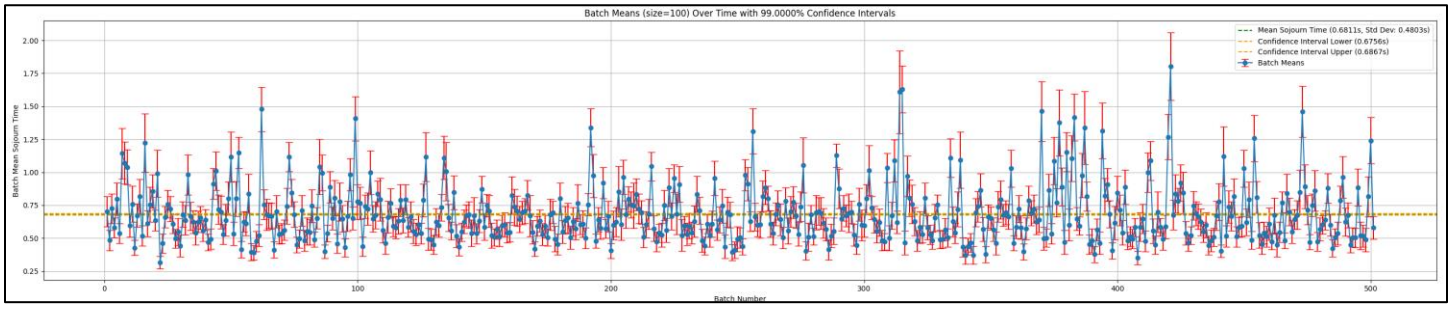


Figure 4: Batch means sojourn times when μ= 8.0 jobs/seconds

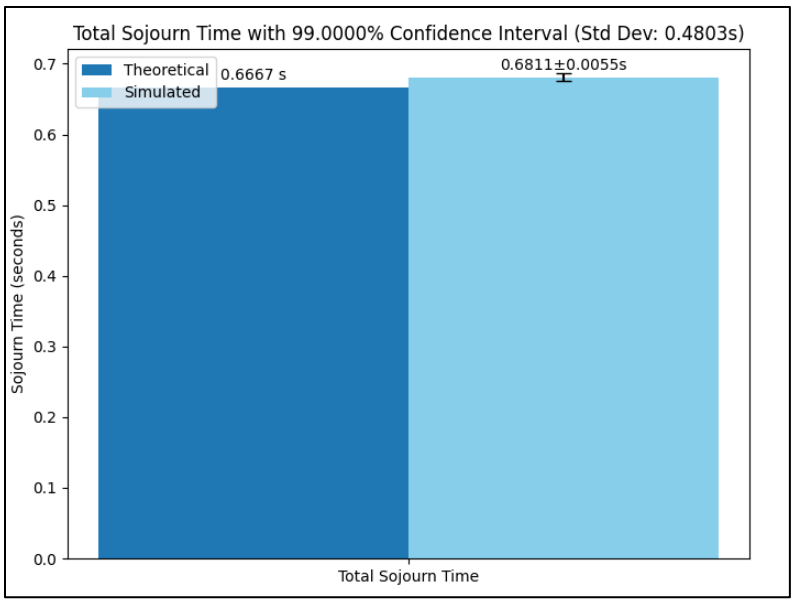


Figure 5: Comparison of simulated vs theoretical time

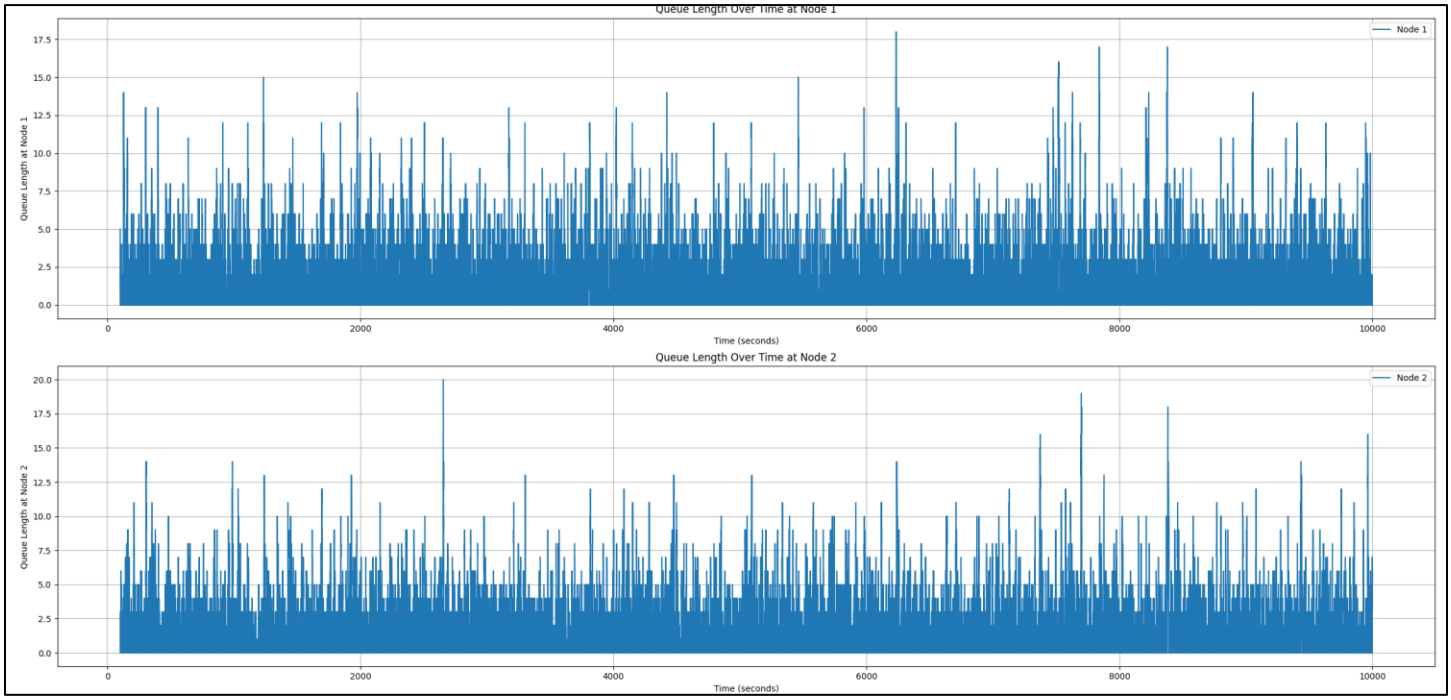


Figure 6: Queue lengths at node 1 and node 2 with time when μ =8.0

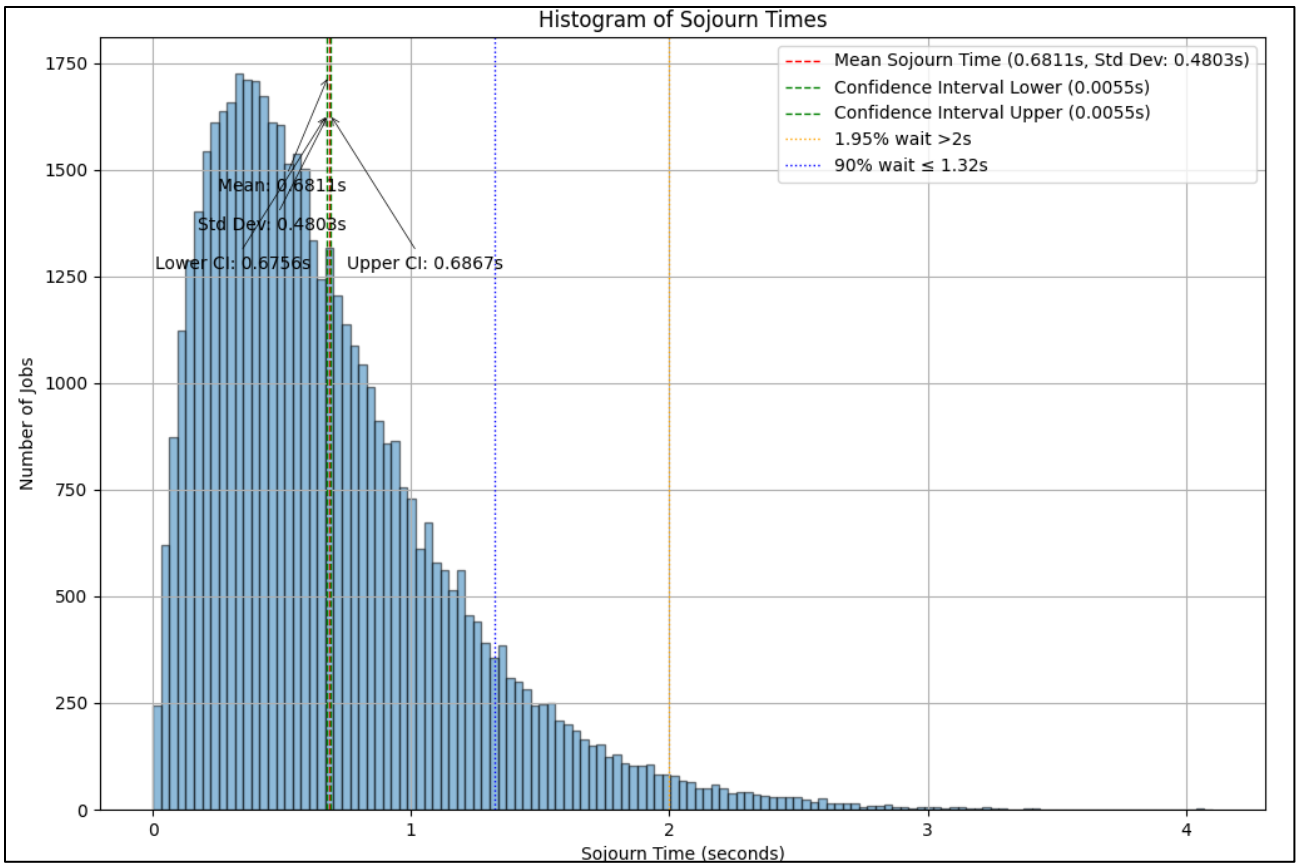


Figure 7: Number of Jobs vs Sojourn times when μ =8.0

Program Output:

```
PS D:\517> python .\QNetSim.py

Running simulation with service rate μ = 4.0
Node 1 still busy at time 10000.038196229216
Node 2 still busy at time 10000.038196229216
Mean Sojourn Time: 1031.4893 ± 7.6279 seconds (Std Dev: 588.4992 seconds)
Utilization of Node 1: 1.0000
Utilization of Node 2: 0.9842
----------------------------------------

Running simulation with service rate μ = 5.0
Node 1 still busy at time 10000.016955984896
Node 2 still busy at time 10000.016955984896
Mean Sojourn Time: 41.6480 ± 0.1527 seconds (Std Dev: 13.2007 seconds)
Utilization of Node 1: 0.9960
Utilization of Node 2: 0.9921
----------------------------------------

Running simulation with service rate μ = 6.0
Node 1 still busy at time 10000.06666452969
Node 2 still busy at time 10000.06666452969
Mean Sojourn Time: 1.9572 ± 0.0167 seconds (Std Dev: 1.4471 seconds)
Utilization of Node 1: 0.8287
Utilization of Node 2: 0.8286
----------------------------------------

Running simulation with service rate μ = 7.0
Mean Sojourn Time: 0.9991 ± 0.0081 seconds (Std Dev: 0.7032 seconds)
Utilization of Node 1: 0.7150
Utilization of Node 2: 0.7164
----------------------------------------

Running simulation with service rate μ = 8.0
Node 1 still busy at time 10000.058308198906
Node 2 still busy at time 10000.058308198906
Mean Sojourn Time: 0.6811 ± 0.0055 seconds (Std Dev: 0.4803 seconds)
Utilization of Node 1: 0.6277
Utilization of Node 2: 0.6309
----------------------------------------

Running simulation with service rate μ = 9.0
Node 1 still busy at time 10000.009979200375
Node 2 still busy at time 10000.009979200375
Mean Sojourn Time: 0.5065 ± 0.0041 seconds (Std Dev: 0.3558 seconds)
Utilization of Node 1: 0.5544
Utilization of Node 2: 0.5594
----------------------------------------

Running simulation with service rate μ = 10.0
Node 1 still busy at time 10000.062956903748
Node 2 still busy at time 10000.062956903748
Mean Sojourn Time: 0.3985 ± 0.0032 seconds (Std Dev: 0.2770 seconds)
Utilization of Node 1: 0.4969
Utilization of Node 2: 0.4989
----------------------------------------
Simulation finished at time: 10000.0583 seconds
Mean Sojourn Time: 0.6811 ± 0.0055 seconds (Std Dev: 0.4803 seconds) at 99.00% confidence level
Utilization of Node 1: 0.6277
Utilization of Node 2: 0.6309

--- Comparison with Jackson's Theorem ---
Node  1:
  Simulated Average Queue Length: 1.7125 ± 2.1671 at 99.00% confidence level
  Theoretical Utilization: 0.6250
  Simulated Utilization: 0.6277
  Theoretical Average Queue Length: 1.6667
  Simulated Average Queue Length: 1.7125 ± 2.1671 at 99.00% confidence level
  Theoretical Sojourn Time: 0.3333 seconds
Node  2:
  Simulated Average Queue Length: 1.7184 ± 2.1672 at 99.00% confidence level
  Theoretical Utilization: 0.6250
  Simulated Utilization: 0.6309
  Theoretical Average Queue Length: 1.6667
  Simulated Average Queue Length: 1.7184 ± 2.1672 at 99.00% confidence level
  Theoretical Sojourn Time: 0.3333 seconds
For the entire system:
  Total Theoretical Sojourn Time: 0.6667 seconds
  Simulated Total Sojourn Time: 0.6811 ± 0.0055 seconds (Std Dev: 0.4803 seconds) at 99.00% confidence level
  Percentage Error in Sojourn Time: 2.1712%

-------Little's Law (L = λW)-------
Simulated Average Number of Jobs in System: 3.4309 ± 0.0249 (Std Dev: 3.0648)
Calculated L (λW): 3.4057
Difference (λW - Simulated L): -0.0252
Little's L (λW) / Simulated Average Number of Jobs in System: 0.9926
Percentage Error from theoretical value: 0.7410%
Confidence Interval for Mean Sojourn Time: ± 0.0055 at 99.00% confidence level
```

Figure 8 : Program Output for exponential service times simulation run

2. Plots for non-Poisson arrivals and service times:

For running non Poisson simulations, I defined a combination of service and inter-arrival times:-

```python
def run_non_poisson_simulations_and_plot():

    combinations = [
        {
            'name': 'Weibull Arrivals and Erlang Service Times',
            'arrival_distribution': 'weibull',
            'arrival_params': {'shape': 3.0, 'scale': 0.1},
            'service_distribution': 'erlang',
            'service_params': {'k': 2, 'theta': 0.5},

            'lambda_system': 1 / (0.1 * scipy.special.gamma(1 + 1 / 3.0))
        },
        {
            'name': 'Lognormal Arrivals and Hyperexponential Service Times',
            'arrival_distribution': 'lognormal',
            'arrival_params': {'mu': -1.0, 'sigma': 0.5},
            'service_distribution': 'hyperexponential',
            'service_params': {'p': [0.6, 0.4], 'means': [1/4, 1/6]},
            'lambda_system': 1 / np.exp(-1.0 + (0.5 ** 2) / 2)
        },
        {
            'name': 'Hypoexponential Arrivals and Gamma Service Times',
            'arrival_distribution': 'hypoexponential',
            'arrival_params': {'lambdas': [2.0, 2.0, 2.0]},
            'service_distribution': 'gamma',
            'service_params': {'shape': 2.0, 'scale': 0.5},
            'lambda_system': 1 / (1/2.0 + 1/2.0 + 1/2.0)
        }
    ]

    total_jobs = 10000
    warmup_period = 100
    simulation_period = 9900
    seed = 12981
    batch_size = 100
    confidence_level = 0.95
```

a. Weibull Arrivals and Erlang Service Times

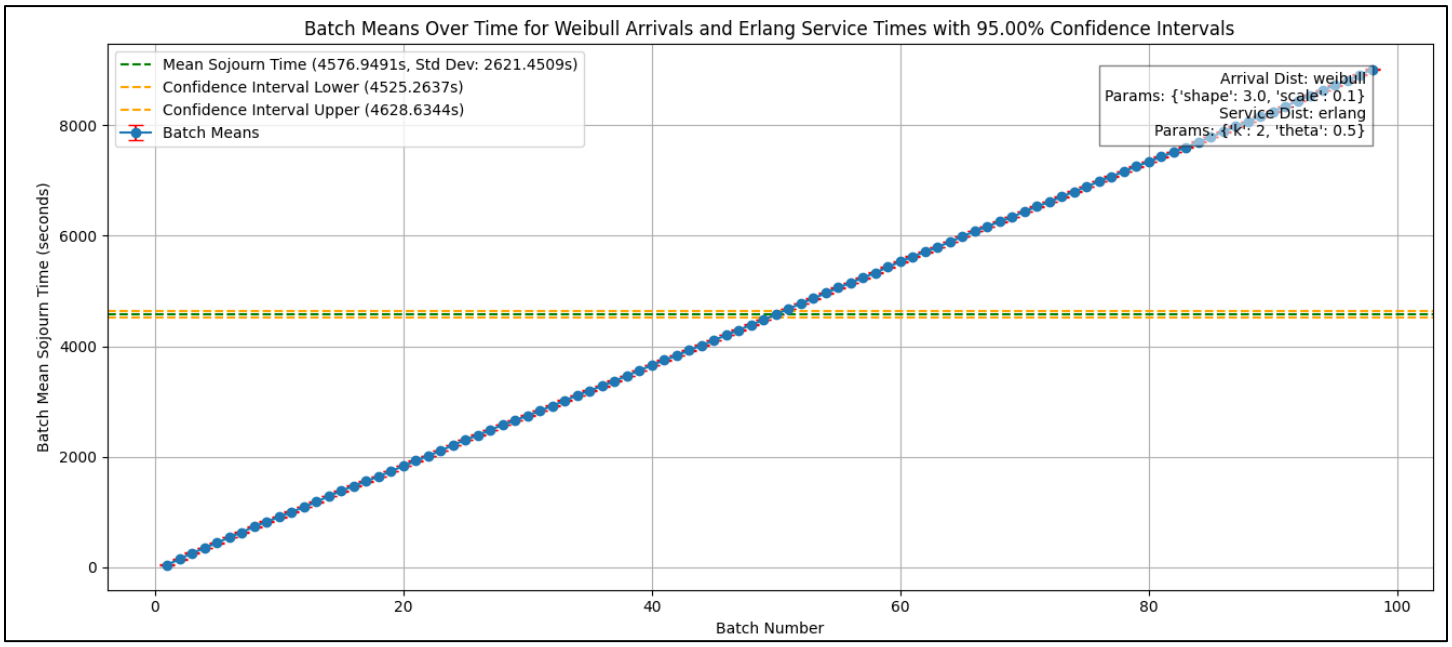


Figure 9: Batch means (batch size = 100) for sojourn times.

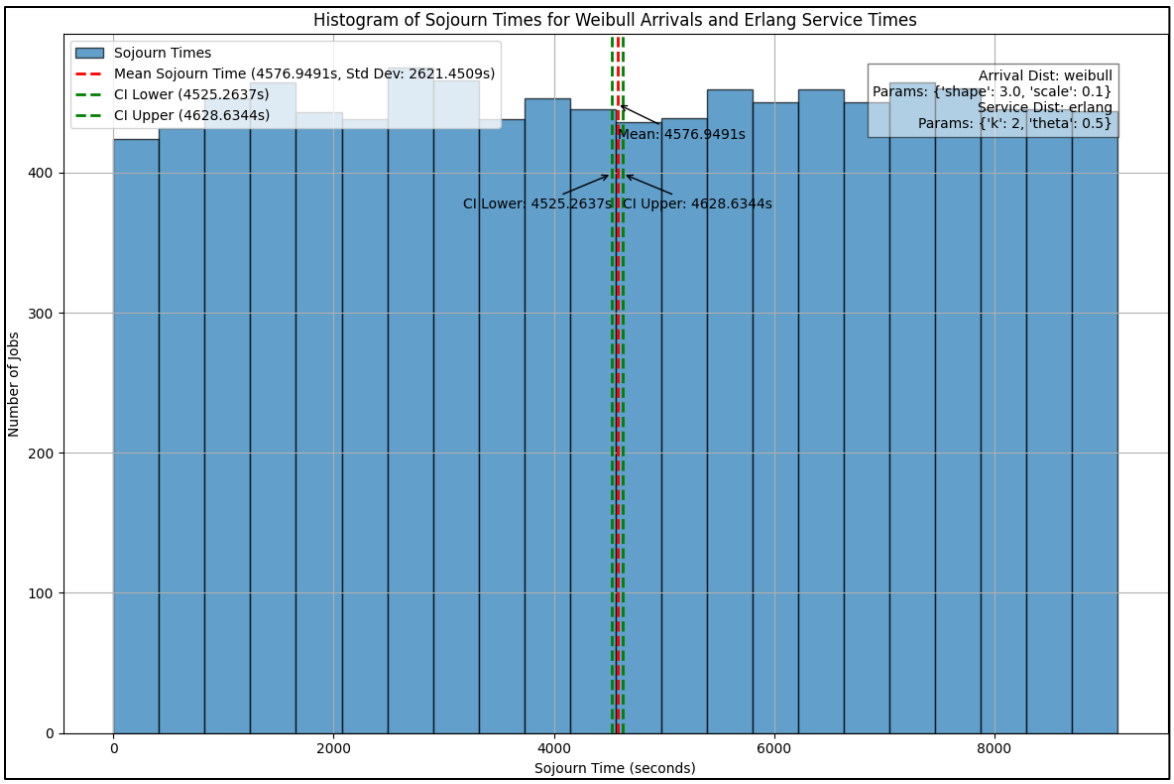


Figure 10: Number of Jobs vs Sojourn times

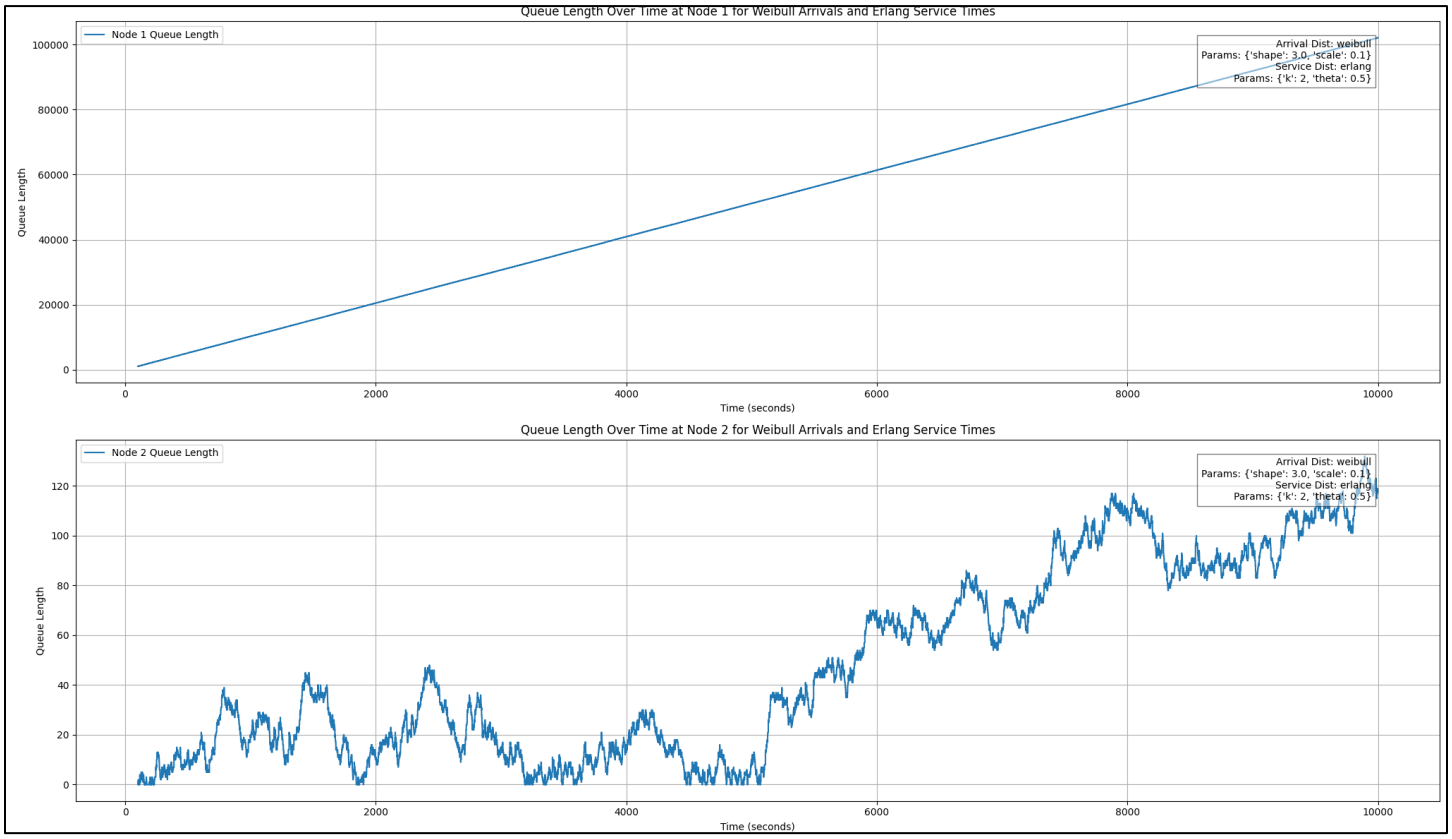


Figure 11: Queue Lengths at different nodes. Unstable system as seen from strictly increasing Node 1 queue length and no steady state at node 2

b. LogNormal Arrivals and HyperExponential Service Times

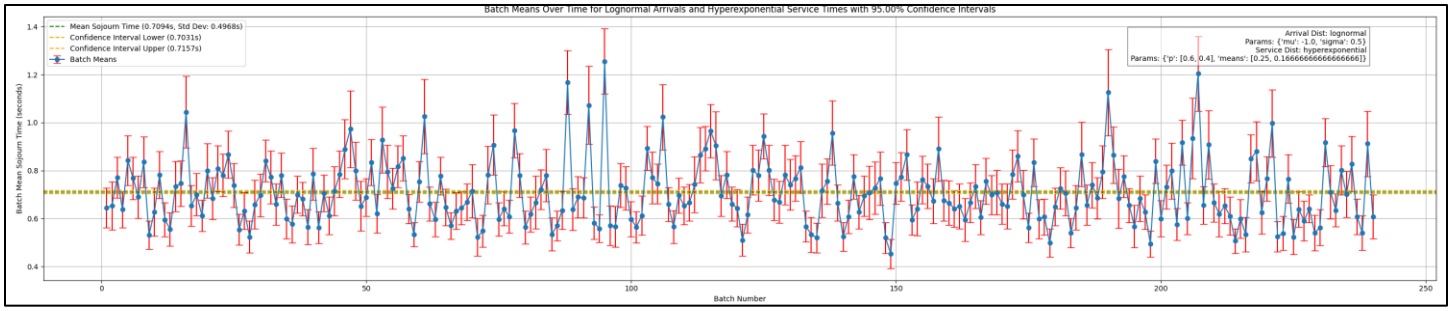


Figure 12:: Batch means (batch size = 100) for sojourn times

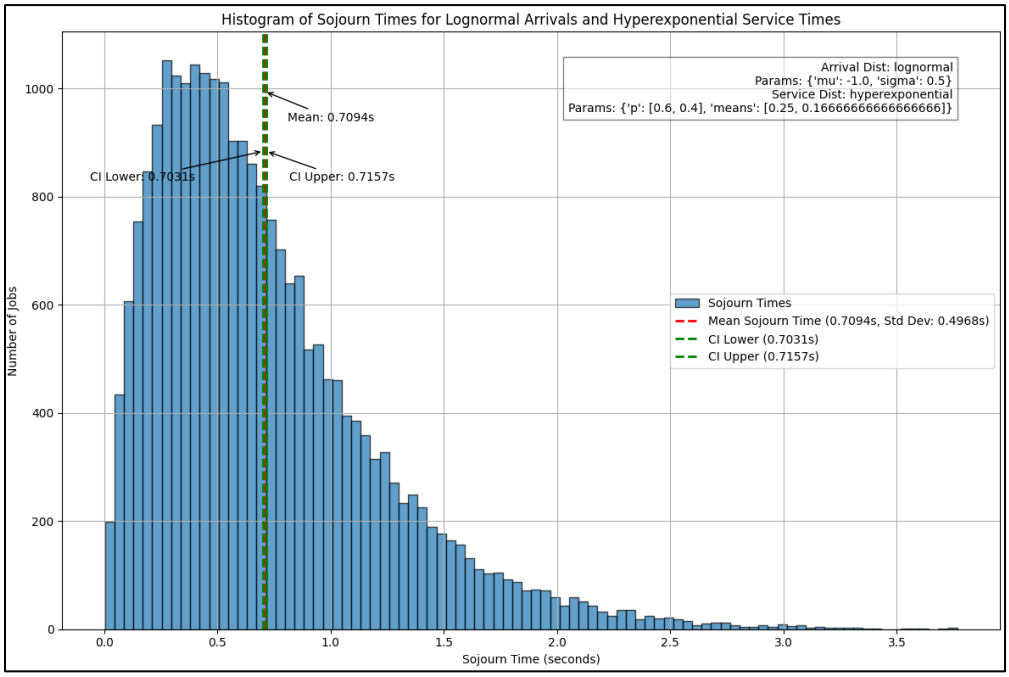


Figure 13: Number of Jobs vs Sojourn times

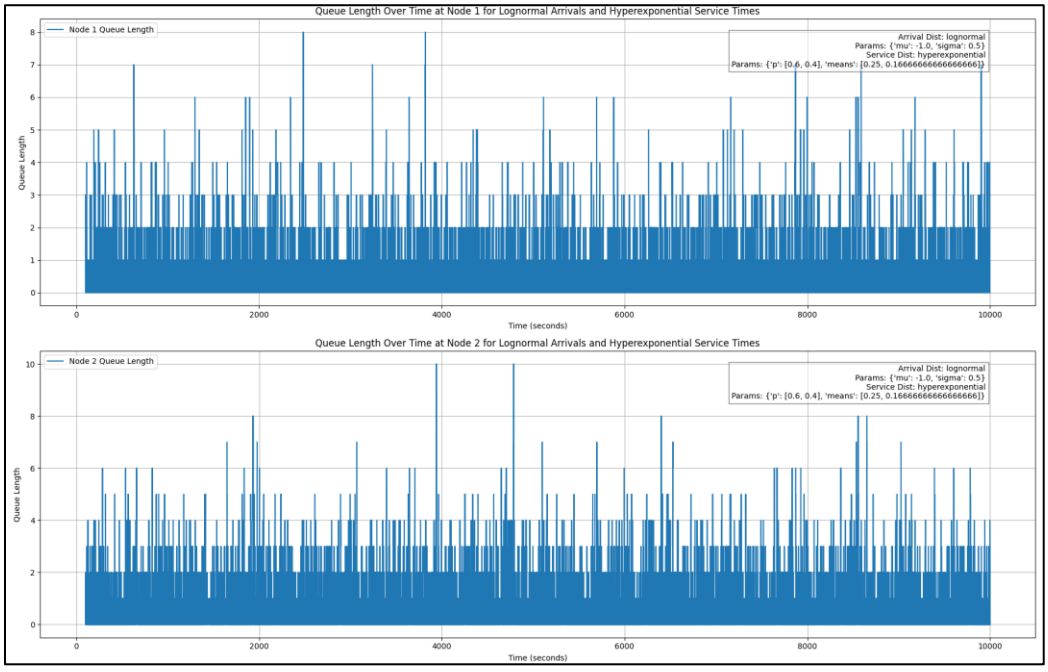


Figure 14: Queue lengths at different nodes

c. HyperExponenetal Arrivals and Gamma Service Times

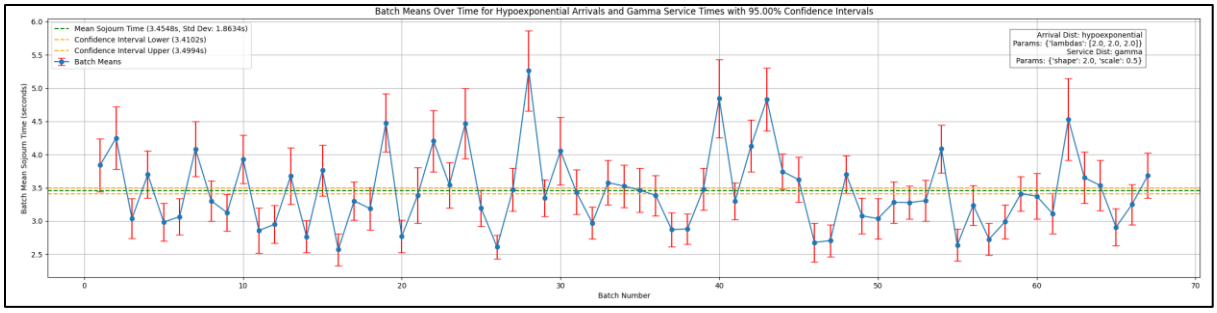


Figure 15:: Batch means (batch size = 100) for sojourn times

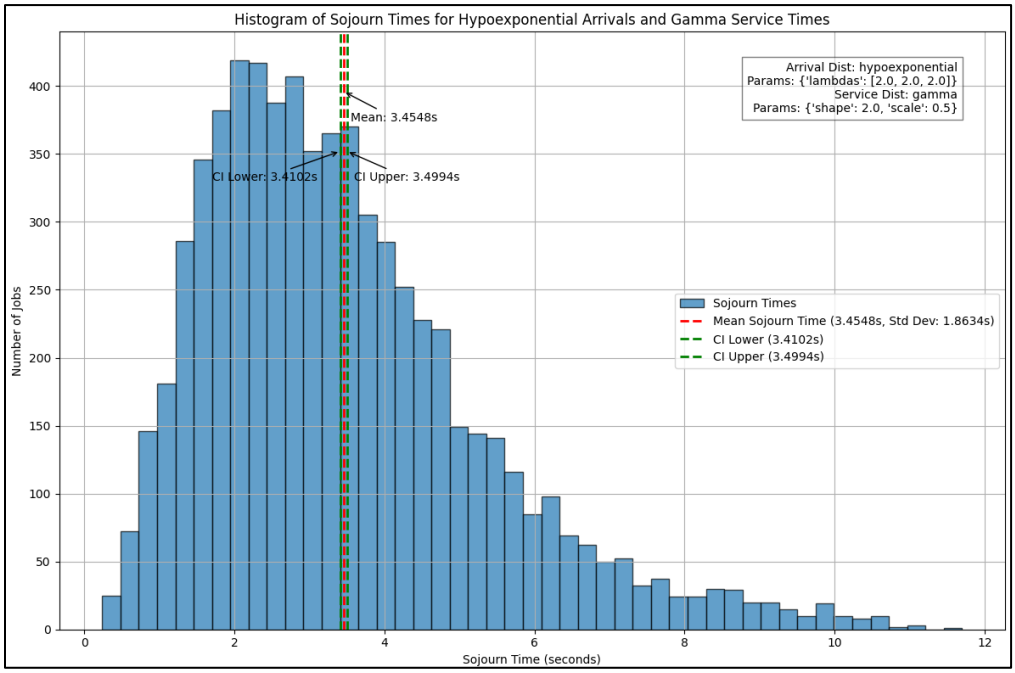


Figure 16: Histogram for Number of Jobs vs Sojourn times for specified distribution

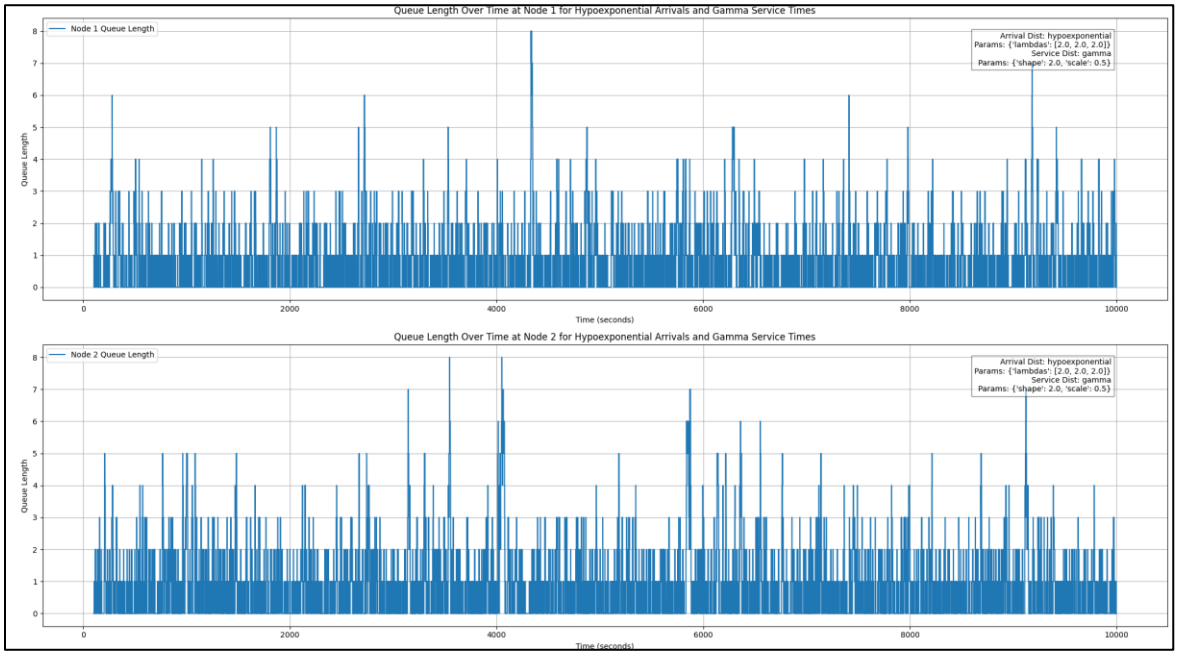


Figure 17: Queue Lengths at node 1 and node 2 for 10000s simulation run

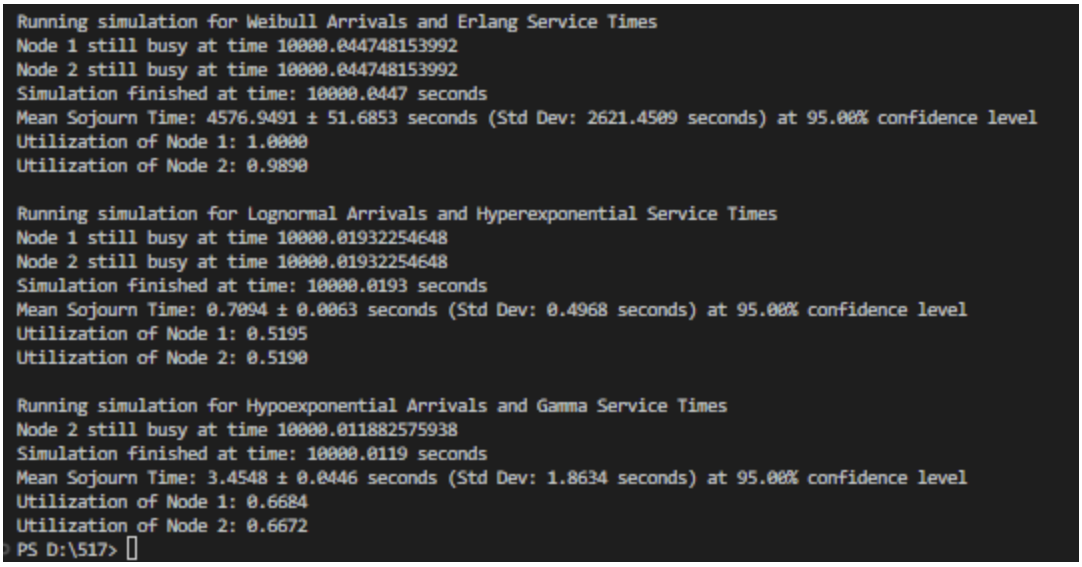


Figure 18: Program Output for running a combination of non-poisson time generation intervals for the simulation

References:

- Cooper, J. C. B. & Applied Probability Trust. (2005). *The poisson and exponential distributions*. Applied Probability Trust. https://neurophysics.ucsd.edu/courses/physics_171/exponential.pdf
- *Event-Driven Simulation of M/M/1 Queues — qmodels 1.0.1 documentation*. (n.d.). https://qmodels.readthedocs.io/en/latest/mm1.html
- Wikipedia contributors. (2023, October 28). *Phase-type distribution*. Wikipedia. https://en.wikipedia.org/wiki/Phase-type_distribution