CSE 517: Simulation of Queueing Networks using an Event Stack.

Nimai Patel

nimaipatel@psu.edu

Code organization and documentation.

Skip to the next section for simulation results.

All the code for this project including the simulation login, event stack, arbitrary phase type random number generators, confidence interval calculation et cetera is written and tested on **Python 3.9** without any dependencies on external libraries. I have used **matplotlib** to plot the charts presented in this document.

The core simulation logic is in the file **queue_simulation.py**.

1. **Simulation** dataclass**:** This object holds the configuration and state for the simulation. Example:

```
s = Simulation(
    DURATION=1000,
    ARRIVAL_DIST=Get_Exponential_Dist(arrival_rate),
    SERVICE_DIST=[
        Get_Exponential_Dist(service_rate_1),
        Get_Exponential_Dist(service_rate_2),
    ],
    LOG_FILE_NAME=LOG_FILE_NAME,
)
```

   DURATION is used to define the units of time for how long the simulation is to run.

   ARRIVAL_DIST expects the random number generator for the distribution of interarrival times.

   SERVICE_DIST expects a list of random number generators for a chain of tandem queues. i.e., the i^{th} entry in the list corresponds to the service rate distribution for the i^{th} queue.

LOG_FILE_NAME is optional and used for logging the events in CSV format.

2. **Simulation_Run**: this is a function that accepts a **Simulation** object and returns the results from the simulation. It returns the queue population distribution for the n queues, the overall system population distribution, and a list of sojourn times. Example:

   ```
   [q1_freq, q2_freq], sys_freq, sojourn_times = Simulation_Run(s)
   ```

3. **Get_Phase_Type_Dist(alphas, S): alpha** is a list of n probabilities, which correspond to the starting probability in the n transient states. **S** is the nxn transient generator matrix. The function returns the random number generator that corresponds to the phase-type distribution corresponding to alpha and S.

4. **Get_Erlang_Dist(k, lam):** returns Erlang-K distribution random number generator with **k** states and rate = **lam**.

5. **Get_Exponential_Dist(lam):** returns exponential distribution random number generator with rate = **lam**.

Verification with Jackson's Theorem

The file **verify_jackson.py** simulates exponential arrivals and two queues with exponential service rates. It then prints the simulation results and prints the calculated expected values for these results using Jackson's theorem and Little's Law. You can modify the means of the exponential service and arrivals to verify for other values.

Output:

```
$ python3 ./verify_jackson.py
Probability Distribution from simualtion:
| Size | Queue 1  | Queue 2  | System   |
| 0    | 0.78055  | 0.81023  | 0.63080  |
| 1    | 0.17362  | 0.15037  | 0.26193  |
| 2    | 0.03594  | 0.03124  | 0.07684  |
| 3    | 0.00725  | 0.00669  | 0.02196  |
| 4    | 0.00252  | 0.00137  | 0.00714  |
| 5    | 0.00012  | 0.00010  | 0.00119  |
| 6    | 0.00000  | 0.00001  | 0.00015  |
| 7    | 0.00000  | 0.00000  | 0.00000  |
|_____________________________________|

Equilibrium State Probability Distribution from Jackson's Theorem:
| Size | Queue 1  | Queue 2  | System   |
| 0    | 0.77778  | 0.80000  | 0.62222  |
| 1    | 0.17284  | 0.16000  | 0.26272  |
| 2    | 0.03841  | 0.03200  | 0.08327  |
| 3    | 0.00854  | 0.00640  | 0.02348  |
| 4    | 0.00190  | 0.00128  | 0.00621  |
| 5    | 0.00042  | 0.00026  | 0.00158  |
| 6    | 0.00009  | 0.00005  | 0.00039  |
| 7    | 0.00002  | 0.00001  | 0.00009  |
| 8    | 0.00000  | 0.00000  | 0.00002  |
| 9    | 0.00000  | 0.00000  | 0.00001  |
| 10   | 0.00000  | 0.00000  | 0.00000  |
|_____________________________________|

Total jobs inbound   = 1951
Total jobs completed = 1949

Average Sojourn Time (expected) = 0.26786s
Average Sojourn Time (measured) = 0.26502s
Standard Deviation              = 0.18694s
95% Confidence interval         = (0.26502 ± 0.00830)s

Average number of jobs in system (expected) = 0.53571
Average number of jobs in system (measured) = 0.51684
Standard Deviation                          = 0.80553
95% Confidence interval                     = (0.51684 ± 0.04993)

Refer event_log.csv for event log
```

This outputs the observed probability distribution for the size of the two queues and the overall system. It also prints the expected equilibrium probability distribution using Jackson's theorem.

Along with this it also returns the average and standard deviation for the number of jobs in the system and the expected value for the average number of jobs in the system.

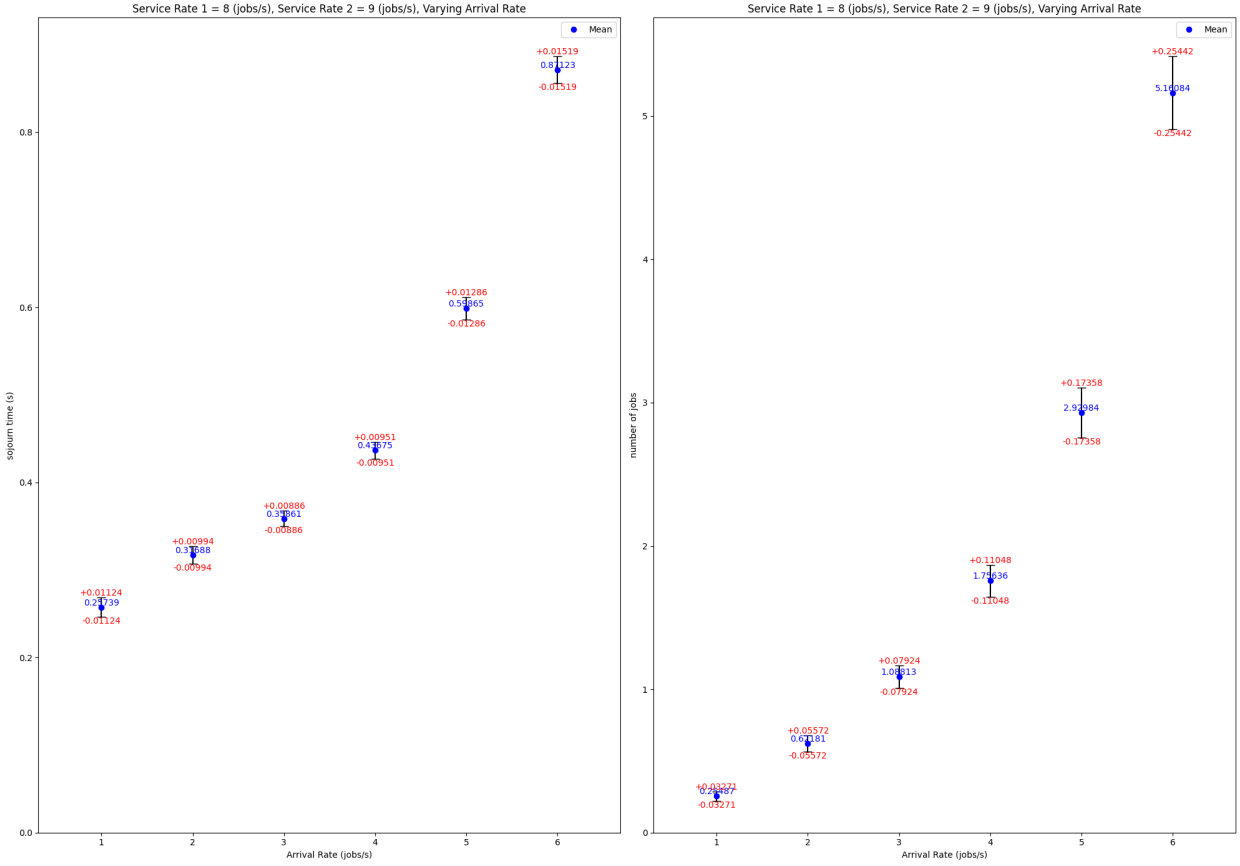
The expected average number of jobs is calculated by finding the expected value of the distribution we found from Jackson’s theorem, which ends up just being the sum of a geometric series.

It also prints the mean and standard deviation for the observed sojourn times of all processed jobs, along with the expected sojourn time. The expected sojourn time is calculated using Little’s law.

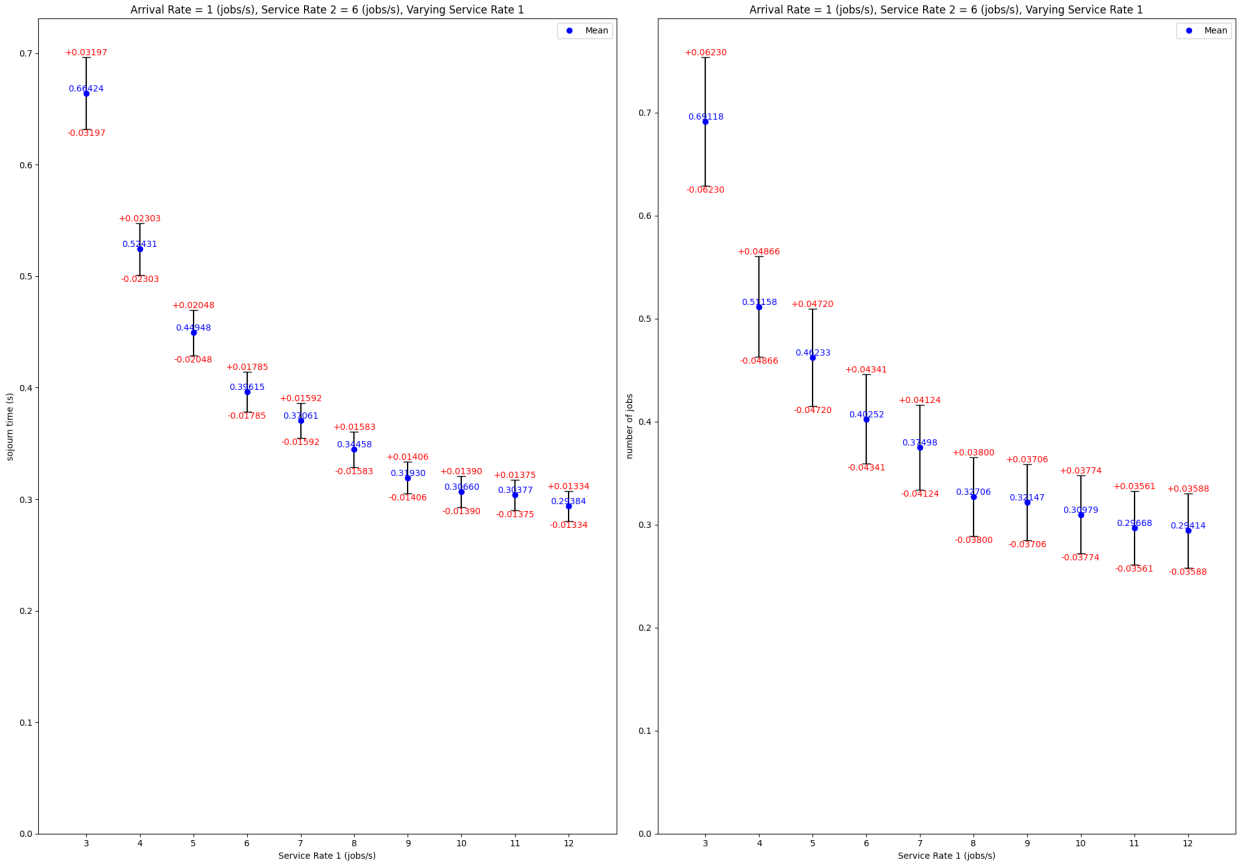
Plots and Results from General Phase-Type Arrivals

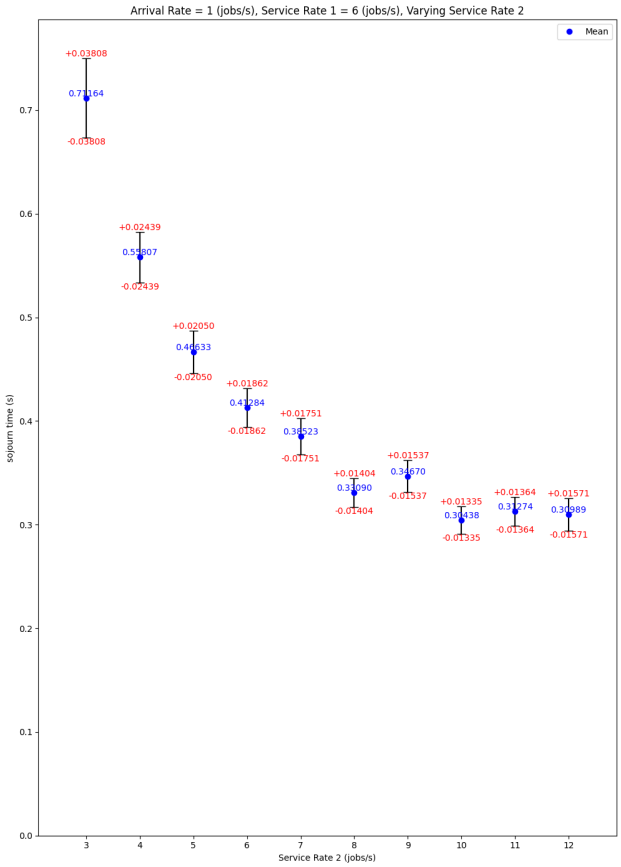
Plots can be reproduced by running **charts.py**. All plots mark the mean with a blue dot and show the 95% confidence interval centered around the mean with red markers.

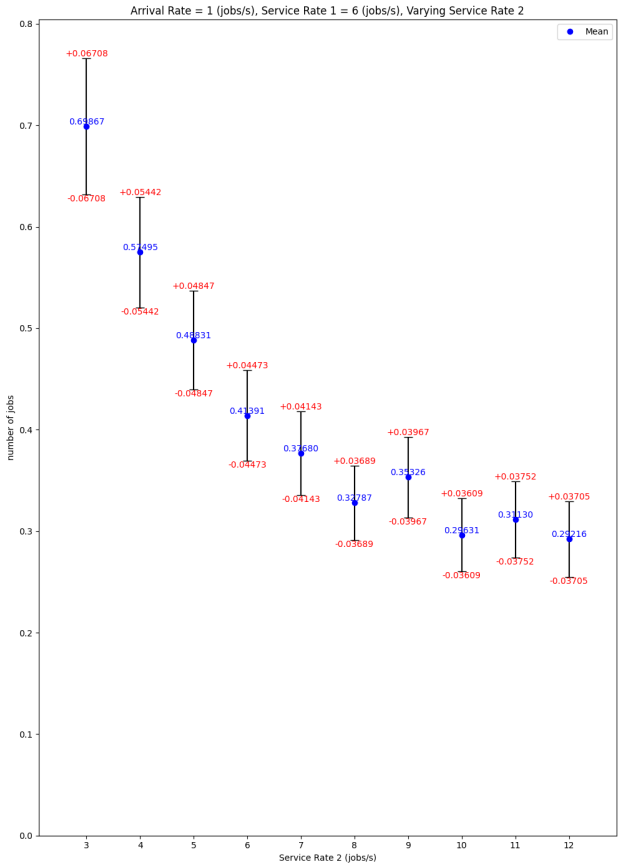
By varying the mean of exponential inter-arrival times, we observe that the system becomes overwhelmed as the mean inter-arrival time approaches the service rate. As arrival rates increase, the graph shows a steep rise, trending toward a vertical line as the system reaches capacity. This illustrates how sensitive the system is to increased arrival rates relative to its service rate.



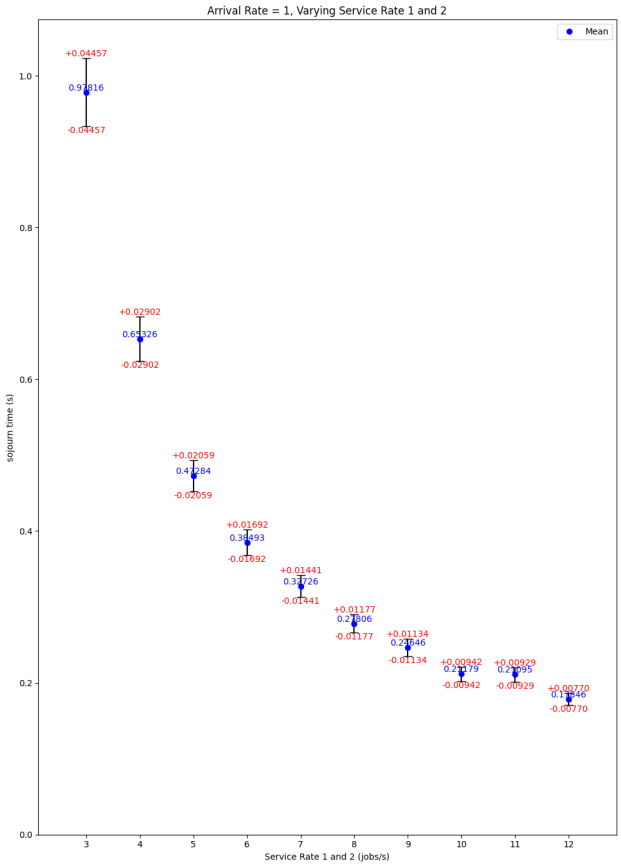
By varying the service rate of one queue while keeping the arrival rate and the service rate of the other queue constant, we observe that the sojourn time initially decreases. However, once the adjusted queue's service rate exceeds that of the other queue, the overall sojourn time plateaus. This indicates that the slower queue becomes the bottleneck, limiting further reductions in sojourn time.

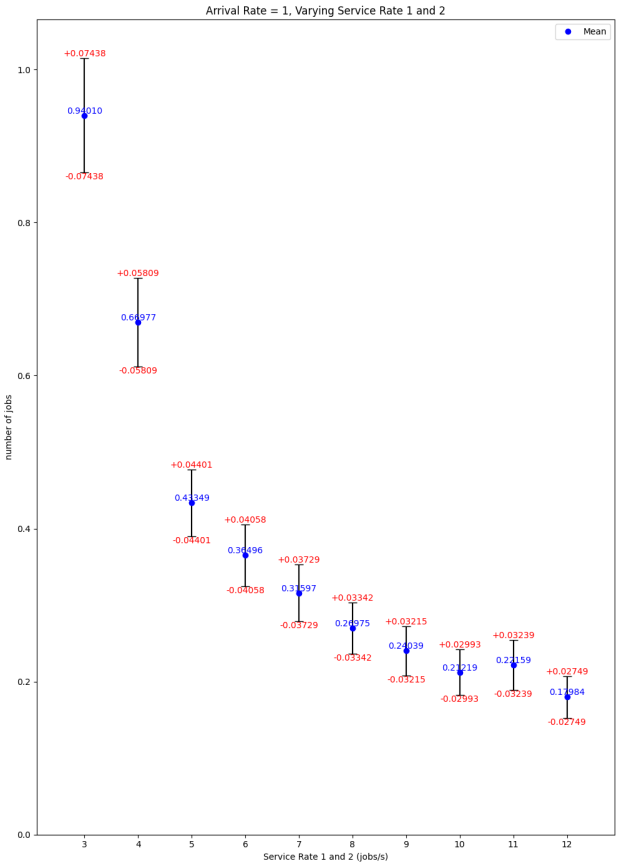



Arrival Rate = 1 (jobs/s), Service Rate 1 = 6 (jobs/s), Varying Service Rate 2
Mean
sojourn time (s)
Service Rate 2 (jobs/s)
+0.03808
0.71164
-0.03808
+0.02439
0.55807
-0.02439
+0.02050
0.46633
-0.02050
+0.01862
0.41784
-0.01862
+0.01751
0.38523
-0.01751
+0.01404
0.32090
-0.01404
+0.01537
0.34670
-0.01537
+0.01335
0.30438
-0.01335
+0.01364
0.31274
-0.01364
+0.01571
0.35989
-0.01571

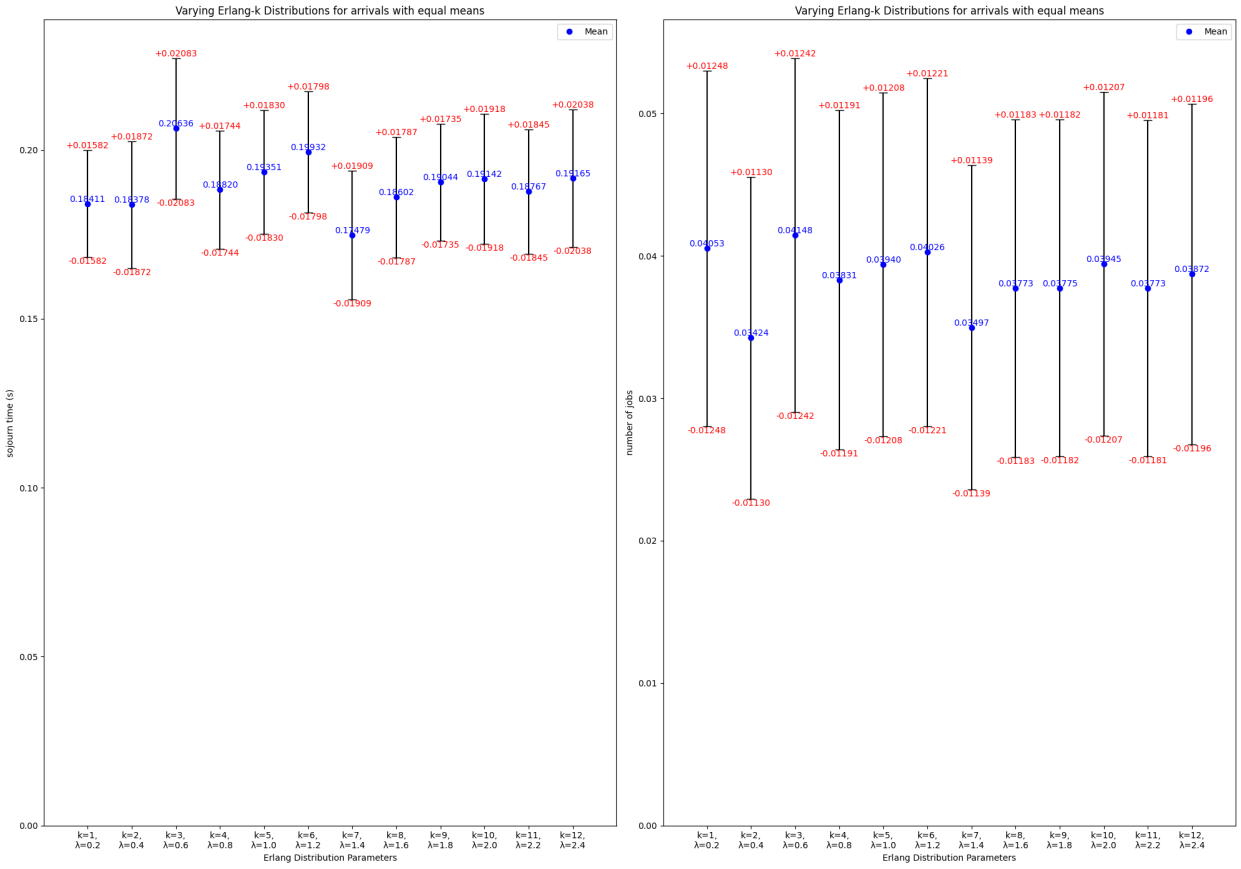

Arrival Rate = 1 (jobs/s), Service Rate 1 = 6 (jobs/s), Varying Service Rate 2
Mean
number of jobs
Service Rate 2 (jobs/s)
+0.06708
0.63867
-0.06708
+0.05442
0.51495
-0.05442
+0.04847
0.48831
-0.04847
+0.04473
0.41391
-0.04473
+0.04143
0.37680
-0.04143
+0.03689
0.33787
-0.03689
+0.03967
0.33926
-0.03967
+0.03609
0.29631
-0.03609
+0.03752
0.31130
-0.03752
+0.03705
0.29216
-0.03705

When both service rates are increased simultaneously, we observe a consistent reduction in sojourn time, approaching zero as expected. This shows that increasing the service rate of both queues helps prevent bottlenecks, allowing the system to handle arrivals more efficiently.





Here, we plot the mean sojourn times and average system population across various Erlang-K distributions with differing inter-arrival times. By adjusting K and λ, we maintain a constant mean arrival rate of 5 jobs per second. The stability of the sojourn times supports our class discussion that the assumption of exponential inter-arrival times is not as impactful to system performance as previously thought. (NOTE: k = 1 is essentially an exponential distribution).



Questions Requiring Written Answers

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 endpoints, 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.

For the more general arrivals, we can let the user specify interarrival times, distributions, and service rates for the different servers and network entry points. This can be a function that the user can implement as well, giving them a way to inject real-world traces for the arrivals as well. My implementation already includes helpers for generating any arbitrary phase-type distribution.

Our simulator can use a graph to represent the queueing network. The vertices can be queues associated with a queue size (optional), service distribution, and servers. The edges will represent the routing probabilities from one queue to the other. (constraint: some of all outgoing edges must be equal to one). If we want a fixed path system, we can simply constrain each vertex to have only one outgoing edge. My simulator already handles a chain of any number of queues.

Each queue can maintain a data structure to hold the sojourn time and queue lengths, which can be used at the end of the simulation to conduct individual queues as well as overall system performance evaluation.

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

Littles formula relates the average arrival rate (λ), average waiting time (W), and average number of jobs in the system (L) using the formula.

$$L = \lambda W$$

I have used this result to validate my experimental results, as shown in my simulation output.

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.

If we have different event stacks, we will have to roll back when we receive an event whose timestamp is lower than already processed events. To rollback, we will maintain a stack of processed events and keep popping and applying their inverse to the event stack till the simulation clock is set to just lower than that of the event we are rolling back for.

Our simulation state machine has an operation for applying an event to its state; now, we will have to implement inverse routines to inverse the effects of a specific event, assuming it has just occurred. This way, it will be possible to have multiple event stacks that partition the network.