# Problem 1 Eval:

In this EVAL assignment we will start to explore how to discover characteristics of incoming workload, we will dive into how to measure the overheads of our systems, and we will also study the properties how queues evolve in response to varying traffic conditions.

**a)** First thing first, let’s try to make sense out of the workload that is coming from the client. You might recall that by invoking the client with various values of the -a and -s parameters significantly impacts the load seen by your server. Now it is time to reverse-engineer the characteristics of that traffic. Start with the following to collect the report of 1,000 packets handled at the server:  
   
 ./server\_mt 2222 & ./client -a 6 -s 10 -n 1000 2222

Now, isolate only the lengths of the requests as they are sent from the client. With that, produce a plot of the distribution of the request lengths you have collected. The distribution plot should have on the x-axis a set of time bins, e.g., from 0 (included) to 0.005 (excluded), from 0.005 (included) to 0.010 (excluded), and so on in steps of 0.005 increments. Given each transaction, look at its length. If it is in the range between 0.005 and 0.010 seconds, it falls in the second bin; if it is in the range between 0.010 and 0.015, it falls in the third bin and so on.  
   
On the y-axis, plot how many requests fall in each bin! But do not plot the raw count. Rather, normalize that value by the total number of requests you are plotting. In this case, 1,000. Hooray! You have produced a distribution plot.

**Steps Taken:**

1. Run the command as normal but redirect the output to a log file as such

| ./build/server\_mt 2222 > server\_output.txt |
| --- |

1. Run the client in a separate terminal:

| ./client -a 6 -s 10 -n 1000 2222 |
| --- |

1. Extract the Request Lengths from the Server Output, Given the server outputs lines in the format:

| R<request\_id>:<sent\_timestamp>,<request\_length>,<receipt\_timestamp>,<completion\_timestamp> |
| --- |

1. We can then use awk (a text-processing tool) to extract the <request\_length> field from each line:

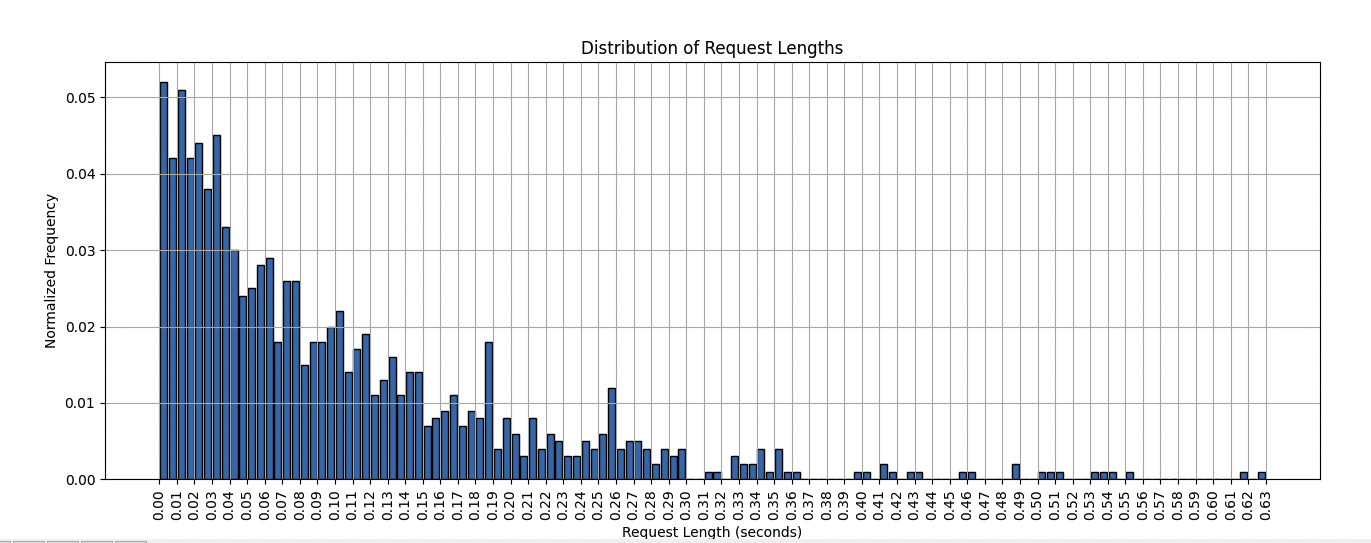
| awk -F '[:,]' '{print $3}' server\_output.txt > request\_lengths.txt |
| --- |

* -F '[:,]': Sets the field separators to : and ,.
* {print $3}: Prints the third field, which corresponds to <request\_length>.

1. After that i cleaned the files to only contain the out I wanted
2. Then I wrote a python script to plot the request lengths

**Script used:**

| import pandas as pd  import matplotlib.pyplot as plt import numpy as np  # read data into a dataframe # Header = None (no header title) # names = ['length'] assigns the column name 'length' to the data df = pd.read\_csv('request\_lengths.txt', header=None, names=['length'])  # define bins max\_length = df['length'].max()  # Create the bins in intervals of 0.05 # np.arange(start, stop, step) # An array of bin edges starting from 0, increasing in steps of 0.005, up to the maximum request length. bins = np.arange(0, max\_length + 0.005, 0.005)  # compute histogram # Counts = the number of data points in each bin # bin\_edges = the edges of the bins counts, bin\_edges = np.histogram(df['length'], bins=bins)  # Normalize the counts normalized\_counts = counts/counts.sum()  # Set x axis ticks ever 0.005 tick\_positions = np.arange(0, max\_length + 0.005, 0.005)  # plotting # Calculates the center of each bin by averaging the left and right edges. bin\_centers = (bin\_edges[:-1] + bin\_edges[1:]) / 2  ''' plt.bar(): Plots a bar chart. bin\_centers: Positions of the bars on the x-axis. normalized\_counts: Heights of the bars. width=0.004: Width of each bar (slightly less than the bin width of 0.005 to avoid overlap). align='center': Centers the bars on the bin\_centers. edgecolor='black': Sets the edge color of the bars. plt.grid(True): Adds a grid to the plot for better readability. plt.show(): Displays the plot. ''' plt.bar(bin\_centers, normalized\_counts, width=0.004, align='center', edgecolor='black')  # Only show ever other tick (each segment will then show 2 bars) plt.xticks(tick\_positions[::2], rotation='vertical') plt.xlabel('Request Length (seconds)') plt.ylabel('Normalized Frequency') plt.title('Distribution of Request Lengths')  plt.grid(True) plt.show()  # print(f"Maximum request length: {df['length'].max()}") # print("Bins:", bins) # print("Counts:", counts) |
| --- |



**b)**  By using the same procedure used in the previous part, produce a distribution plot of the inter-arrival  
   
time between any two subsequent requests. Say that request R0 is sent (look at the sent timestamp!)  
   
arrives at t0 = 10s and R1 is sent at t1 = 15s, then the inter-arrival time between them is tiat = (t1 −t0).

Compute all the 999 inter-arrival times you have observed and plot their distribution just like you did above, except that this time you will normalize by 999.

**Steps Taken:**

1. Run the command as normal but redirect the output to a log file as such

| ./build/server\_mt 2222 > server\_output.txt |
| --- |

1. Run the client in a separate terminal:

| ./client -a 6 -s 10 -n 1000 2222 |
| --- |

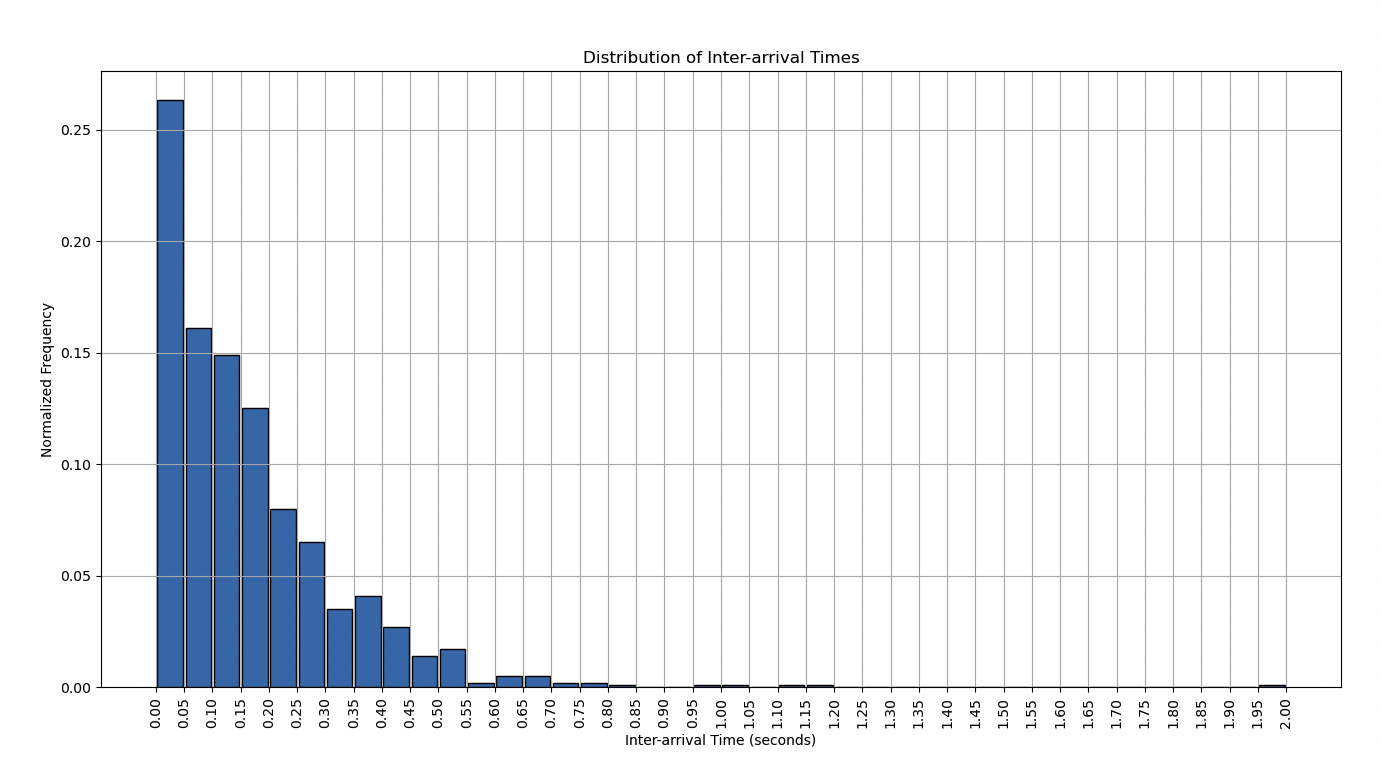
1. Each output is of the form

| R<request\_id>:<sent\_timestamp>,<request\_length>,<receipt\_timestamp>,<completion\_timestamp> |
| --- |

We want to extract the sent\_timestamp and the completion timestamp. We can do this using a python script.

**Script Used:**

| import pandas as pd import numpy as np import matplotlib.pyplot as plt  # Read the server output file containing the request data # Structure: R<request\_id>:<sent\_timestamp>,<request\_length>,<receipt\_timestamp>,<completion\_timestamp> """ sep='[:,]': Defines the separators for the fields, which are either : or , in the server output. engine='python': Specifies the Python engine for parsing, which is more flexible when dealing with complex separators. """ df = pd.read\_csv('server\_output.txt', header=None, names=['request\_id', 'sent\_timestamp', 'request\_length', 'receipt\_timestamp', 'completion\_timestamp'], sep='[:,]', engine = 'python')  # extract the sent timestamp  sent\_timestamps = df['sent\_timestamp'].astype(float)  # Compute the inter-arrival times """ Calculating Inter-arrival Times: sent\_timestamps.diff(): This calculates the difference between consecutive timestamps, effectively giving us the inter-arrival times. For example, if the first timestamp is 10 and the second is 15, the inter-arrival time is 15 - 10 = 5. dropna(): Removes the first result, which is NaN because there's no previous timestamp to subtract from the first entry. After applying diff(), the first row will have a NaN value because theres no prior timestamp to calculate the difference. Now, inter\_arrival\_times contains the differences between each pair of consecutive sent timestamps, giving us the 999 inter-arrival times for 1000 requests. """ inter\_arrival\_times = sent\_timestamps.diff().dropna() # Drop the first NaN value  max\_iat = inter\_arrival\_times.max()  # Set x axis ticks ever 0.005 tick\_positions = np.arange(0, max\_iat + 0.05, 0.05)   # Define bins for the inter-arrival times, just like for request lengths bins = np.arange(0, max\_iat+0.05, 0.05) # Adjust bin size as necessary  # Compute the histogram """ Calculating the Histogram: np.histogram(inter\_arrival\_times, bins=bins): Computes the histogram of the inter-arrival times. inter\_arrival\_times: The data to be binned. bins=bins: The bin edges we defined earlier. counts: An array representing how many inter-arrival times fall into each bin. bin\_edges: The edges of the bins used to group the inter-arrival times. """ counts, bin\_edges = np.histogram(inter\_arrival\_times, bins=bins)  normalized\_counts = counts / 999 """ Calculating the Bin Centers: bin\_edges[:-1]: All the bin edges except the last one. bin\_edges[1:]: All the bin edges except the first one. (bin\_edges[:-1] + bin\_edges[1:]) / 2: Averages the left and right bin edges to get the center of each bin. This is useful for plotting the bar chart where the bars are centered on the bins. """ bin\_centers = (bin\_edges[:-1] + bin\_edges[1:])/2  # plt.figure(figsize=(16, 8)): Creates a figure with a larger size (16 units wide and 8 units tall). plt.figure(figsize=(16,8))  """ plt.bar(bin\_centers, normalized\_counts, width=0.004, align='center', edgecolor='black'): Plots a bar chart with the bin centers on the x-axis and the normalized counts on the y-axis. width=0.045: Sets the width of the bars (slightly less than the bin width to avoid overlap). align='center': Ensures the bars are centered on the bin centers. edgecolor='black': Adds a black outline to the bars for better visibility. """ plt.bar(bin\_centers, normalized\_counts, width=0.045, align='center', edgecolor='black')  plt.xticks(tick\_positions, rotation='vertical')  plt.xlabel('Inter-arrival Time (seconds)') plt.ylabel('Normalized Frequency') plt.title('Distribution of Inter-arrival Times')  plt.grid(True) plt.show() |
| --- |



**c)** Time to reverse-engineer things! Let’s start from the distribution of request lengths. Use your favorite programming language to generate 10,000 samples from the following theoretical distributions:

(1) A Normal distribution with mean 1/10 and standard deviation 1;

(2) An Exponential distribution with mean 1/10;

(3) A uniform distribution with mean 1/10.

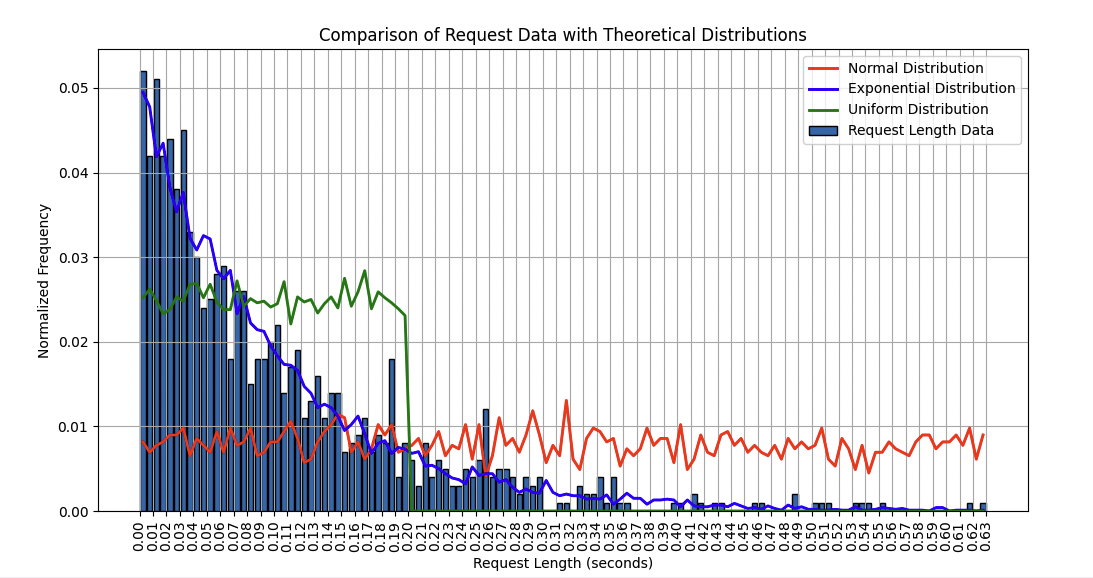
Plot these distribution together (on the same plot, just different lines) with the distribution you previously acquired from your server run. Thus, your plot should have a total of 4 different lines (I suggest having lines instead of bars for this plot) in it. Then, comment on which ones of the curves matches more closely with the experimental data. If there is one of them that matches remarkably close, you have successfully reverse-engineered the characteristics of your input load!

**Steps Taken:**

The script is essentially the same as 1a, except this time I used np to create normal, exponential and uniform distributions as per the requirements. The script is below and is commented.

**Script Used:**

| import pandas as pd  import matplotlib.pyplot as plt import numpy as np  ################################################ Code from 1a # read data into a dataframe # Header = None (no header title) # names = ['length'] assigns the column name 'length' to the data df = pd.read\_csv('request\_lengths.txt', header=None, names=['length'])  # define bins max\_length = df['length'].max()  # Create the bins in intervals of 0.05 # np.arange(start, stop, step) # An array of bin edges starting from 0, increasing in steps of 0.005, up to the maximum request length. bins = np.arange(0, max\_length + 0.005, 0.005)  # compute histogram # Counts = the number of data points in each bin # bin\_edges = the edges of the bins counts, bin\_edges = np.histogram(df['length'], bins=bins)  # Normalize the counts normalized\_counts = counts/counts.sum()  # Set x axis ticks ever 0.005 tick\_positions = np.arange(0, max\_length + 0.005, 0.005)  # plotting # Calculates the center of each bin by averaging the left and right edges. bin\_centers = (bin\_edges[:-1] + bin\_edges[1:]) / 2  ################################################ New Code  # Define the number of samples for theoretical distributions num\_samples = 10000 bin\_width = bin\_edges[1] - bin\_edges[0] # Get the bin width (0.005)  # generate samples from normal dist """ Generate samples from the Normal distribution: np.random.normal() generates random samples from a normal distribution. loc=mean\_normal sets the mean of the distribution (1/10). scale=std\_normal sets the standard deviation (1). size=num\_samples specifies the number of samples to generate (10,000). This creates a set of 10,000 random samples from a normal distribution with a mean of 1/10 and standard deviation of 1. """ mean\_normal = 1/10 std\_normal = 1 normal\_samples = np.random.normal(loc=mean\_normal, scale=std\_normal, size=num\_samples)  # Generate samples from the exponential dist """ Generate samples from the Exponential distribution: np.random.exponential() generates random samples from an exponential distribution. scale=mean\_exponential sets the scale parameter, which is the inverse of the rate parameter λ.  Since the mean of an exponential distribution is 1/λ, setting the scale to  1/10 ensures a mean of 1/10. This creates 10,000 samples from an exponential distribution with a mean of 1/10. """ mean\_exponential = 1/10 exponential\_samples = np.random.exponential(scale=mean\_exponential, size=num\_samples)  # Generate samples from the uniform dist """ Generate samples from the Uniform distribution: np.random.uniform() generates random samples from a uniform distribution. low=low\_uniform sets the lower bound of the distribution (0). high=high\_uniform sets the upper bound of the distribution (2/10), ensuring that the mean is centered around 1/10. This creates 10,000 samples uniformly distributed between 0 and 2/10. """ low\_uniform = 0 # to center around mean 1/10 high\_uniform = 2/10 # to center around mean 2/10 uniform\_samples = np.random.uniform(low=low\_uniform, high=high\_uniform, size=num\_samples)  # Compute the histograms for th theoretical distributions """ Compute histograms for the theoretical distributions: np.histogram() is used to calculate the histogram of each distribution (Normal, Exponential, and Uniform). bins=bins ensures that all histograms use the same bins as the experimental data, so they can be compared directly. density=True normalizes the histogram so that the area under the curve sums to 1, making it a probability density function (PDF) rather than a frequency count. """ normal\_counts, \_ = np.histogram(normal\_samples, bins=bins, density=True) exponential\_counts, \_ = np.histogram(exponential\_samples, bins=bins, density=True) uniform\_counts, \_ = np.histogram(uniform\_samples, bins=bins, density=True)  # Scale the PDFs by the bin width to match the scale of the normalized data normal\_counts \*= bin\_width exponential\_counts \*= bin\_width uniform\_counts \*= bin\_width  # Plot the request length data as bars plt.figure(figsize=(12, 6)) # Increase figure size for better readability  """ Plot the experimental data: plt.bar() is used to plot the experimental data as bars. bin\_centers: The positions of the bars on the x-axis (centered on each bin). normalized\_counts: The height of each bar, representing the normalized frequency. width=0.004: The width of each bar is slightly smaller than the bin size to prevent overlap. alpha=0.6: Sets the transparency of the bars, so that the lines can be more visible. """ plt.bar(bin\_centers, normalized\_counts, width=0.004, align='center', edgecolor='black', label='Request Length Data')  # Plot the theoretical distributions as lines plt.plot(bin\_centers, normal\_counts, label='Normal Distribution', color='red', linewidth=2) plt.plot(bin\_centers, exponential\_counts, label='Exponential Distribution', color='blue', linewidth=2) plt.plot(bin\_centers, uniform\_counts, label='Uniform Distribution', color='green', linewidth=2)  # Only show ever other tick (each segment will then show 2 bars) plt.xticks(tick\_positions[::2], rotation='vertical') plt.xlabel('Request Length (seconds)') plt.ylabel('Normalized Frequency') plt.title('Comparison of Request Data with Theoretical Distributions') # Show the legend clearly plt.legend(loc='best') plt.grid(True) plt.show() |
| --- |



We can clearly see that the request length data distribution seems to line up perfectly with an exponential distribution.

**d)** Do the same with the inter-arrival times. But this time, compare it with the following three references: (1) A Normal distribution with mean 1/6 and standard deviation 1;  
 (2) An Exponential distribution with mean 1/6;  
 (3) A uniform distribution with mean 1/6.

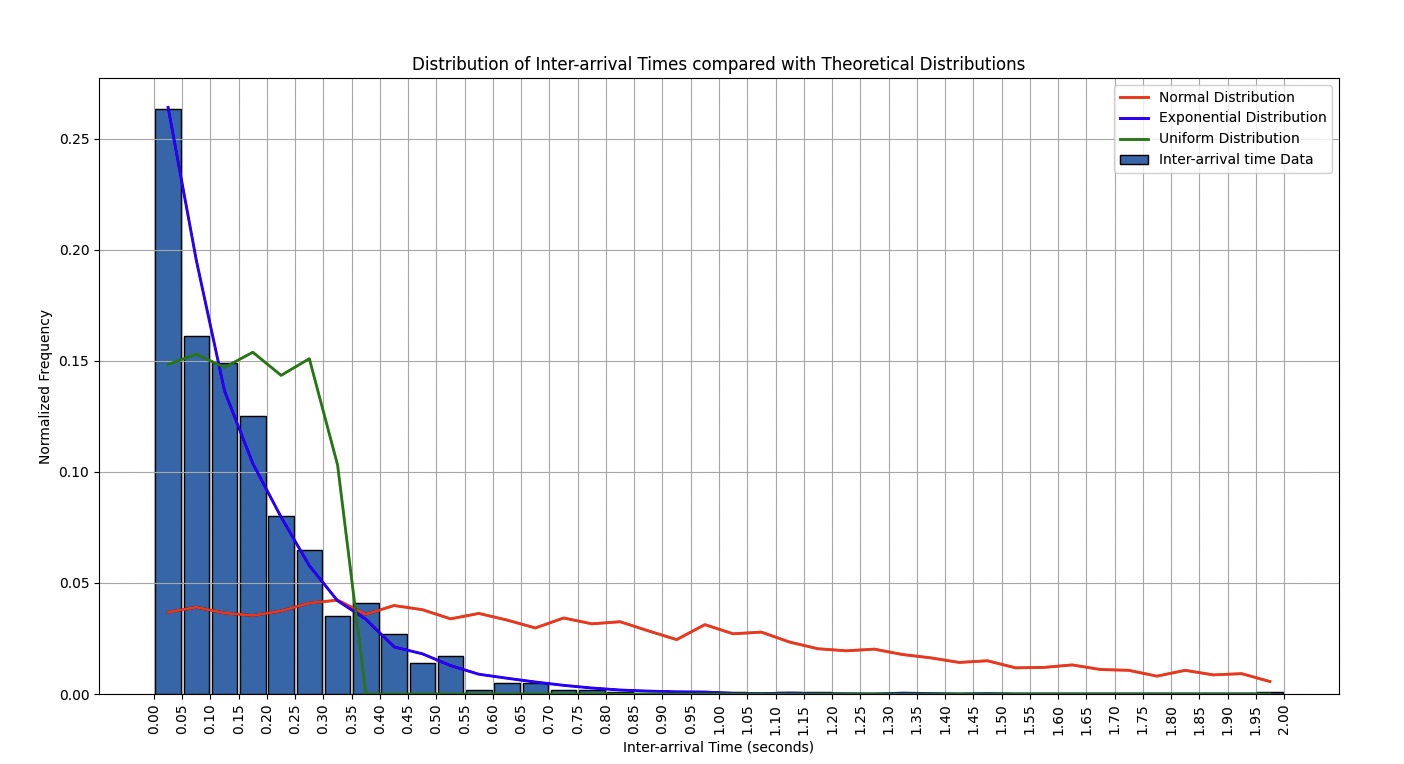
Produce the comparison plot and comment on the match between the experimental and theoretical curves. At this point, can you tell me what the -a and -s parameters control, exactly?

**Steps Taken:**

Pretty much the same steps taken as in 1c but this time interweaved with the code from 1b.

**Script Used:**

| import pandas as pd import numpy as np import matplotlib.pyplot as plt  ################################################### Code from 1b # Read the server output file containing the request data # Structure: R<request\_id>:<sent\_timestamp>,<request\_length>,<receipt\_timestamp>,<completion\_timestamp>  df = pd.read\_csv('server\_output.txt', header=None, names=['request\_id', 'sent\_timestamp', 'request\_length', 'receipt\_timestamp', 'completion\_timestamp'], sep='[:,]', engine = 'python')  # extract the sent timestamp  sent\_timestamps = df['sent\_timestamp'].astype(float)  # Compute the inter-arrival times  inter\_arrival\_times = sent\_timestamps.diff().dropna() # Drop the first NaN value max\_iat = inter\_arrival\_times.max()  # Set x axis ticks ever 0.005 tick\_positions = np.arange(0, max\_iat + 0.05, 0.05)   # Define bins for the inter-arrival times, just like for request lengths bins = np.arange(0, max\_iat+0.05, 0.05) # Adjust bin size as necessary  # Compute the histogram counts, bin\_edges = np.histogram(inter\_arrival\_times, bins=bins)  normalized\_counts = counts / 999 bin\_centers = (bin\_edges[:-1] + bin\_edges[1:])/2  ################################################### New Code  # Define the number of samples for theoretical distributions num\_samples = 10000 bin\_width = bin\_edges[1] - bin\_edges[0] # Get the bin width (0.05)  # generate samples from normal dist  mean\_normal = 1/6 std\_normal = 1 normal\_samples = np.random.normal(loc=mean\_normal, scale=std\_normal, size=num\_samples)  # Generate samples from the exponential dist  mean\_exponential = 1/6 exponential\_samples = np.random.exponential(scale=mean\_exponential, size=num\_samples)  # Generate samples from the uniform dist  low\_uniform = 0 # to center around mean 1/6 high\_uniform = 2/6 # to center around mean 2/6 uniform\_samples = np.random.uniform(low=low\_uniform, high=high\_uniform, size=num\_samples)  # Compute the histograms for th theoretical distributions  normal\_counts, \_ = np.histogram(normal\_samples, bins=bins, density=True) exponential\_counts, \_ = np.histogram(exponential\_samples, bins=bins, density=True) uniform\_counts, \_ = np.histogram(uniform\_samples, bins=bins, density=True)  # Scale the PDFs by the bin width to match the scale of the normalized data normal\_counts \*= bin\_width exponential\_counts \*= bin\_width uniform\_counts \*= bin\_width  # plt.figure(figsize=(16, 8)): Creates a figure with a larger size (16 units wide and 8 units tall). plt.figure(figsize=(16,8)) plt.bar(bin\_centers, normalized\_counts, width=0.045, align='center', edgecolor='black', label='Inter-arrival time Data')  # Plot the theoretical distributions as lines plt.plot(bin\_centers, normal\_counts, label='Normal Distribution', color='red', linewidth=2) plt.plot(bin\_centers, exponential\_counts, label='Exponential Distribution', color='blue', linewidth=2) plt.plot(bin\_centers, uniform\_counts, label='Uniform Distribution', color='green', linewidth=2)  plt.xticks(tick\_positions, rotation='vertical')  plt.xlabel('Inter-arrival Time (seconds)') plt.ylabel('Normalized Frequency') plt.title('Distribution of Inter-arrival Times compared with Theoretical Distributions') # Show the legend clearly plt.legend(loc='best') plt.grid(True) plt.show() |
| --- |

 So from part C we can see that the **Parameter -s** seems to control the mean of the **exponential distribution** of service times. Increasing -s will shift the mean of the service time distribution, resulting in longer average request lengths.

From Part D, we can see that **Parameter -a** likely controls the mean of the **exponential distribution** for inter-arrival times. -a reflects how frequently requests are arriving, and increasing -a would result in shorter inter-arrival times (i.e., more frequent requests).

The **exponential distribution** of the inter-arrival times suggests that the system operates like a **Poisson process**, where requests arrive randomly, but with a fixed average arrival rate.

# Problem 2 Eval:

**a)** First thing first, learn how to take a good queue size average. A good queue average measurement should consider the amount of time the queue remains in a certain state, i.e., it should be a timed average of the queue length.

Let us make an example. Say that your queue at t = 0 has 3 elements in it, and it stays that way until time t = 9 sec, at which point the queue becomes empty. If you do not consider time, the average size would be q = (3 + 0)/2 = 1.5. But this is incorrect: most of the time we see the queue with 3 elements, and only at the end with 0. So an average of 1.5 seems wrong.

The right way to take the average is by weighting the queue state by the time it stays in that state. Thus, the right way to calculate q in our example is q = 3·(9/10 )+0·(1/10 ) = 2.7. You can see how this

is a better average of queue size over a 10 seconds time window starting from time t = 0.  
 Now, measure the queue length for the case where your queue-enabled server is invoked with the

following parameters:

./server\_q 2222 & ./client -a 14 -s 15 -n 1000 2222

Use the queue snapshots produced by the worker thread to measure the queue size as it was observed after each request was observed. Use the time elapsed between two subsequent queue snapshots to weigh that size towards the total average.

### **Steps to Solve 2a:**

1. **Collect Queue Snapshots:** After each request is processed, capture the current queue size.
2. **Compute Durations:** Calculate the time intervals between consecutive snapshots.
3. **Calculate Weighted Average:** Multiply each queue size by the duration it persisted and sum these values. Then, divide by the total time to get the average.

Redirect the server's output to a log file for easier processing:  
  
./server\_q 2222 > server\_log.txt 2>&1 & ./client -a 14 -s 15 -n 1000 2222

1. **Parse the Server Log:**
   * Extract timestamps and queue sizes from the log.
   * Identify lines that indicate request completions (R lines) and queue statuses (Q lines).
2. **Calculate the Time-Weighted Average:**
   * For each queue snapshot, determine how long the queue stayed in that state before the next snapshot.
   * Multiply each queue size by its corresponding duration.
   * Sum all these products and divide by the total time to obtain the average.
3. **Implement a Python Script:**
   * Automate the parsing and calculation process using Python.
   * The script should read the log file, extract relevant data and perform calculations.

Server log format:

R<request ID>:<sent timestamp>,<request length>,<receipt timestamp>,<start timestamp>,<completion timestamp>

Q:[R<request ID>,R<request ID>,...]

**Parsing the server log**

| import re  def parse\_server\_log(log\_file\_path):   """  Parses the server log and extracts completion timestamps  and queue sizes.   returns a list of tuples: (completion\_timestamp, queue\_size)  """   pattern\_r = re.compile(r'^R\d+:(\d+\.\d+),(\d+\.\d+),(\d+\.\d+),(\d+\.\d+),(\d+\.\d+)$')  pattern\_q = re.compile(r'^Q:\[(.\*?)\]$')   data = []  current\_completion\_time = None   with open(log\_file\_path, 'r') as file:  for line in file:  line = line.strip()  # Check for R line  match\_r = pattern\_r.match(line)   if match\_r:  # Extract completion time (5th value)  completion\_time = float(match\_r.group(5))  current\_completion\_time = completion\_time  continue # move to the next line to find Q   """  Logic Here for match\_q:   Non-Empty Queue (Q:[R1,R2,R3]):  queue\_contents = match\_q.group(1)  queue\_contents = "R1,R2,R3"  queue\_contents.strip() == ''  "R1,R2,R3".strip() → "R1,R2,R3"  "R1,R2,R3" == '' → False  Else Block:  queue\_size = queue\_contents.count('R') → 3 (R1, R2, R3)  """  match\_q = pattern\_q.match(line)  if match\_q and current\_completion\_time is not None:  # Extract queue size by counting requests in Q  queue\_contents = match\_q.group(1)  if (queue\_contents.strip() == ""):  queue\_size = 0  else:  # Count number of 'R' entries  queue\_size = queue\_contents.count("R")  # Append the tuple  data.append((current\_completion\_time, queue\_size))  current\_completion\_time = None # Reset for next R  continue  return data |
| --- |

**Explanation:**

* **Purpose:** Extracts relevant data (completion timestamps and corresponding queue sizes) from the server log.
* **Regular Expressions:**
  + **pattern\_r:**
    - Matches lines that indicate request completions.
    - ^R\d+:: Line starts with R followed by one or more digits and a colon.
    - (\d+\.\d+): Captures floating-point numbers (timestamps and lengths).
    - There are five capture groups corresponding to the five fields in the R line.
  + **pattern\_q:**
    - Matches lines that indicate the current queue status.
    - ^Q:\[(.\*?)\]$: Line starts with Q:[, captures any characters inside the brackets, and ends with ].
* **Process:**
  + **Matching R Lines:**
    - Uses pattern\_r to identify lines that represent request completions.
    - Extracts the **completion timestamp** (5th capture group).
    - Stores it in current\_completion\_time.
    - Continues to the next line to find the corresponding Q line.
  + **Matching Q Lines:**
    - Uses pattern\_q to identify lines that represent queue statuses.
    - Ensures that there's a current\_completion\_time with which to associate the queue size.
    - Extracts the queue contents inside the brackets.
    - Determines the **queue size** by counting the number of R entries:
      * If the queue is empty (Q:[]), queue\_size = 0.
      * Otherwise, counts how many R entries are present.
    - Appends the tuple (completion\_timestamp, queue\_size) to data.
    - Resets current\_completion\_time for the next iteration.
* **Return Value:**
  + A list of tuples where each tuple contains:
    - **completion\_timestamp**: When a request was completed.
    - **queue\_size**: The number of requests in the queue immediately after processing that request.

**Calculating the Time-Weighted Average**

| def compute\_time\_weighted\_average(data):  """  Computes the time-weighted average queue size.  'data' is a list of tuples: (completion\_timestamp, queue\_size)  """   if not data:  return 0.0    total\_time\_weighted\_queue = 0.0  total\_time = 0.0   for i in range (len(data)):  current\_time, current\_queue = data[i]  if i < len(data) - 1:  # more than 2 entries left  next\_time = data[i+1][0]  duration = next\_time - current\_time  else:  # for the last 2 entries, assume the duration is the same as previous  if len(data) >= 2:  duration = data[i][0] = data[I-1][0]  else:  duration = 0 # last data point   if duration < 0:  print(f"Warning: Negative duration between {current\_time} and {next\_time}. Skipping")  continue  total\_time\_weighted\_queue += duration \* current\_queue  total\_time += duration   if total\_time == 0:  return 0.0    average = total\_time\_weighted\_queue/total\_time   return average |
| --- |

**Explanation:**

* **Purpose:** Calculates the time-weighted average queue size using the extracted data.
* **Process:**
  + **Check for Empty Data:**
    - If data is empty, returns 0.0 as there's nothing to calculate.
  + **Iterate Through Each Data Entry:**
    - **Loop Index i:** Ranges from 0 to len(data) - 1.
    - **current\_time and current\_queue:** Extracted from the current tuple.
  + **Determine Duration:**
    - **If Not Last Entry (i < len(data) - 1):**
      * **next\_time:** Timestamp of the next entry.
      * **duration:** Time difference between next\_time and current\_time.
    - **If Last Entry:**
      * **Assumption:** Duration is the same as the previous interval.
      * **duration:** Calculated as the difference between the last and second-last timestamps.
      * **Edge Case:** If there's only one data point, duration is set to 0.
  + **Handle Negative Durations:**
    - If duration is negative (which shouldn't happen if timestamps are in order), print a warning and skip this interval.
  + **Accumulate Weighted Queue Size and Total Time:**
    - **total\_time\_weighted\_queue += current\_queue \* duration**
    - **total\_time += duration**
  + **Calculate Average:**
    - **If total\_time is 0:** Return 0.0 to avoid division by zero.
    - **Else:** average = total\_time\_weighted\_queue / total\_time
* **Return Value:**
  + The calculated time-weighted average queue size as a floating-point number.

**Running the Script:**

| def main():  log\_file = './2a\_server\_log.txt'  data = parse\_server\_log(log\_file)  average\_queue\_size = compute\_time\_weighted\_average(data)  print(f"Time-Weighted Average Queue Size: {average\_queue\_size:.4f}")  if \_\_name\_\_ == "\_\_main\_\_":  main() |
| --- |

**Output:**

Time-Weighted Average Queue Size: 8.1203

**b)** Now let us repeat the computation of the queue size average as in Q1a but this time sweep through the -a parameter passed to the server. In particular, run the first experiment for a value of 1; then a second time with a value of 2; and so on until and including the case where the value is 15. Thus, you will run 15 experiments in total. This might take a while, so try to automate the runs and dump the results into a file for later analysis.

By reusing what you learned in hw1, also extract utilization and response time averages from each of the 15 experiments. Now, you should have three sets of 15 values each: (1) utilization, (2) average response time, (3) average queue length.

Finally, produce a plot that depicts the trend of the average response time (on the y-axis as line 1) and average queue size (on the y-axis as line 2) as a function of the server utilization x-axis. What relationship do you discover between how response time and queue length averages evolve as a result of increasing utilization?

Steps taken:

Basically its an integration of stuff used from HW1 and problem a.

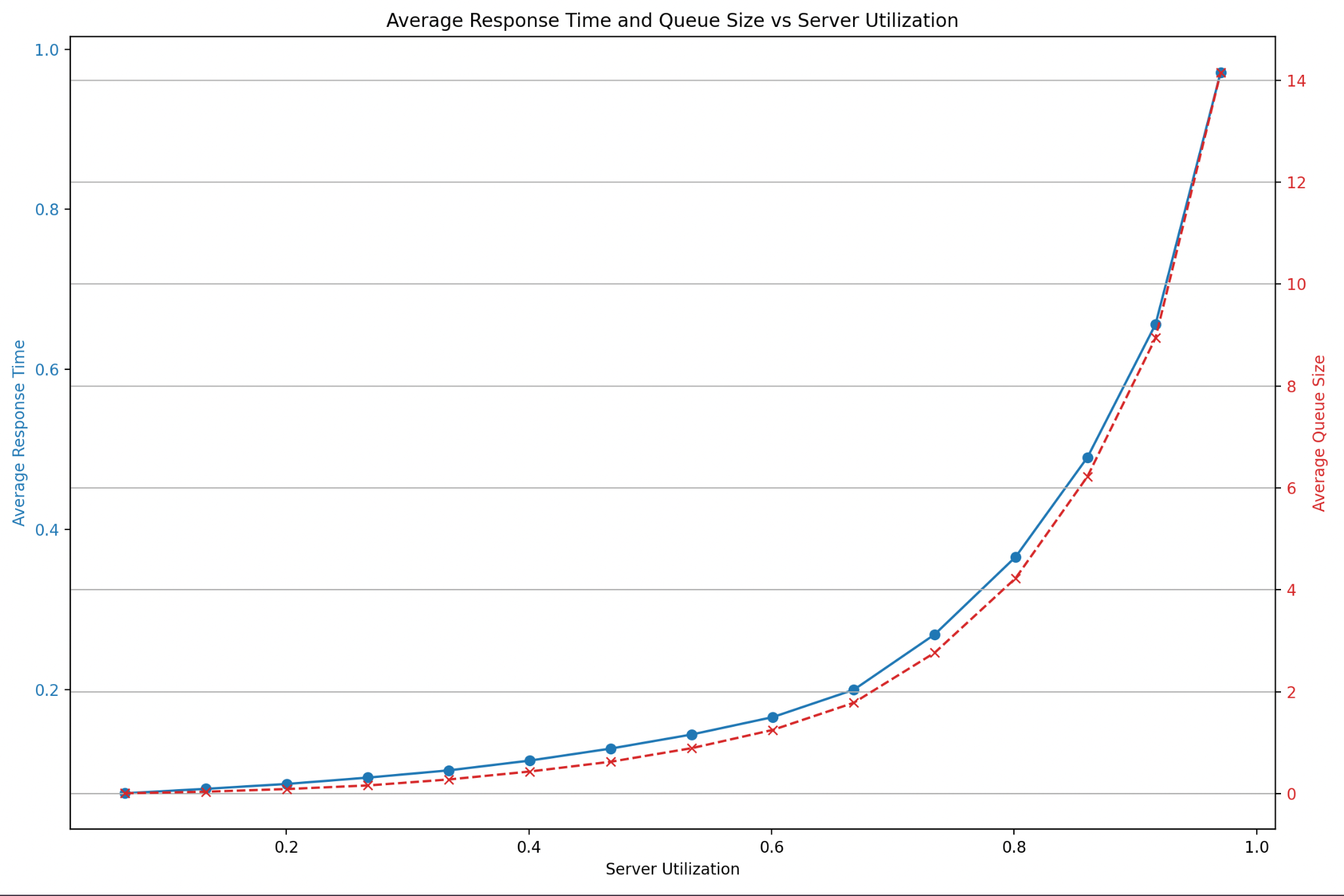
The code analyzes how varying the -a parameter (arrival rate) affects server performance. I first ran the 15 experiments manually and saved the results to a log folder.The script then for each experiment, it extracts key metrics: average queue size, server utilization, and average response time. The data is then processed and plotted to observe how these metrics change as the server utilization varies. Specifically:

1. **Log Parsing**: Reads the server logs to extract request and queue details.
2. **Metrics Calculation**:
   * **Average Queue Size**: Computes the time-weighted average size of the queue during the experiment.
   * **Server Utilization**: Calculates the proportion of time the server is busy processing requests.
   * **Average Response Time**: Measures the time taken to complete requests from receipt to completion.
3. **Plotting Results**: Visualizes the trends of average response time and queue size as a function of server utilization to discover their relationship.

**Script Used:**

| import re import pandas as pd import matplotlib.pyplot as plt  def read\_server\_logs(log\_file\_path):  """  Reads and parses the server log file to extract request information and queue sizes.   Args:  log\_file\_path (str): Path to the server log file.   Returns:  pd.DataFrame: DataFrame containing request details and corresponding queue sizes.  """  # Define regex patterns for request and queue lines  request\_pattern = re.compile(r'^R(\d+):(\d+\.\d+),(\d+\.\d+),(\d+\.\d+),(\d+\.\d+),(\d+\.\d+)$')  queue\_pattern = re.compile(r'^Q:\[([R\d,]\*)\]$')   data = []   with open(log\_file\_path, 'r') as file:  for line in file:  line = line.strip()   # Match request lines  req\_match = request\_pattern.match(line)  if req\_match:  req\_id = int(req\_match.group(1))  sent\_ts = float(req\_match.group(2))  req\_len = float(req\_match.group(3))  recv\_ts = float(req\_match.group(4))  start\_ts = float(req\_match.group(5))  comp\_ts = float(req\_match.group(6))  data.append({  'req\_id': req\_id,  'sent\_ts': sent\_ts,  'req\_len': req\_len,  'recv\_ts': recv\_ts,  'start\_ts': start\_ts,  'comp\_ts': comp\_ts,  'queue\_size': 0 # Placeholder for queue size  })  continue # Proceed to next line   # Match queue lines  q\_match = queue\_pattern.match(line)  if q\_match and data:  queue\_contents = q\_match.group(1)  if queue\_contents.strip() == "":  queue\_size = 0  else:  # Count number of 'R' entries indicating requests in queue  queue\_size = queue\_contents.count('R')  # Assign queue size to the most recent request  data[-1]['queue\_size'] = queue\_size   return pd.DataFrame(data)  def compute\_weighted\_average\_queue\_size(df):  """  Computes the time-weighted average queue size based on request timestamps.   Args:  df (pd.DataFrame): DataFrame containing request details and queue sizes.   Returns:  float: Time-weighted average queue size.  """  if df.empty:  return 0.0   total\_time\_weighted\_queue = 0.0  total\_time = 0.0   for i in range(len(df)):  current\_time = df['start\_ts'].iloc[i]  current\_queue = df['queue\_size'].iloc[i]   if i < len(df) - 1:  next\_time = df['start\_ts'].iloc[i + 1]  duration = next\_time - current\_time  else:  if len(df) >= 2:  duration = current\_time - df['start\_ts'].iloc[i - 1]  else:  duration = 0 # No duration for single entry   if duration < 0:  print(f"Warning: Negative duration between {current\_time} and {next\_time}. Skipping.")  continue   total\_time\_weighted\_queue += duration \* current\_queue  total\_time += duration   if total\_time == 0:  return 0.0   average = total\_time\_weighted\_queue / total\_time  return average  def calculate\_server\_utilization(df):  """  Calculates the server utilization based on busy time and total experiment duration.   Args:  df (pd.DataFrame): DataFrame containing request details.   Returns:  float: Server utilization ratio.  """  if df.empty:  return 0.0   total\_busy\_time = (df['comp\_ts'] - df['start\_ts']).sum()  experiment\_start = df['recv\_ts'].min()  experiment\_end = df['comp\_ts'].max()  experiment\_duration = experiment\_end - experiment\_start   if experiment\_duration <= 0:  return 0.0   utilization = total\_busy\_time / experiment\_duration  return utilization  def calculate\_average\_response\_time(df):  """  Calculates the average response time for all requests.   Args:  df (pd.DataFrame): DataFrame containing request details.   Returns:  float: Average response time in seconds.  """  if df.empty:  return 0.0   df['response\_time'] = df['comp\_ts'] - df['recv\_ts']  average\_response\_time = df['response\_time'].mean()  return average\_response\_time  def main():  # Initialize lists to store metrics for each experiment  utilization\_values = []  average\_response\_times = []  average\_queue\_sizes = []   # Run experiments for a parameter sweep from 1 to 15  for a in range(1, 16):  log\_file = f'./logs/server\_a{a}.txt'   print(f"Processing log file: {log\_file}")   # Parse the server log  df = read\_server\_logs(log\_file)   # Compute metrics  avg\_queue\_size = compute\_weighted\_average\_queue\_size(df)  utilization = calculate\_server\_utilization(df)  avg\_response\_time = calculate\_average\_response\_time(df)   # Store metrics for plotting  utilization\_values.append(utilization)  average\_queue\_sizes.append(avg\_queue\_size)  average\_response\_times.append(avg\_response\_time)   # Output results for the current experiment  print(f"Results for a={a}:")  print(f" Average Queue Size: {avg\_queue\_size:.4f}")  print(f" Server Utilization: {utilization:.4f}")  print(f" Average Response Time: {avg\_response\_time:.4f} seconds\n")   # Create a DataFrame for plotting  results\_df = pd.DataFrame({  'Utilization': utilization\_values,  'Average Response Time': average\_response\_times,  'Average Queue Size': average\_queue\_sizes  })   # Save results to a CSV file for later analysis  results\_df.to\_csv('./results/experiment\_results.csv', index=False)  print("Experiment results saved to './results/experiment\_results.csv'.")   fig, ax1 = plt.subplots(figsize=(12, 8))   color = 'tab:blue'  ax1.set\_xlabel('Server Utilization')  ax1.set\_ylabel('Average Response Time', color=color)  ax1.plot(results\_df['Utilization'], results\_df['Average Response Time'], label='Average Response Time', marker='o', linestyle='-', color=color)  ax1.tick\_params(axis='y', labelcolor=color)   ax2 = ax1.twinx() # Instantiate a second axes that shares the same x-axis   color = 'tab:red'  ax2.set\_ylabel('Average Queue Size', color=color)  ax2.plot(results\_df['Utilization'], results\_df['Average Queue Size'], label='Average Queue Size', marker='x', linestyle='--', color=color)  ax2.tick\_params(axis='y', labelcolor=color)   fig.tight\_layout()  plt.title('Average Response Time and Queue Size vs Server Utilization')  plt.grid(True)  plt.tight\_layout()  plt.show()   if \_\_name\_\_ == "\_\_main\_\_":  main() |
| --- |

**Output:**



The plot reveals a strong relationship between server utilization, average response time, and average queue size. As **server utilization** increases, both the **average response time** and **average queue size** rise significantly. At low utilization levels, response time and queue size grow slowly. However, as utilization approaches 100%, both metrics increase exponentially.

**c)** By looking at the plot produced above, can you conclude that there is some fixed proportional rela- tionship between queue length and response time? Is there something in the theory covered so far capable of modeling this relationship?

No, I dont think we can conclude there is a fixed proportional relationship between queue length and response time. The plot indicates that both metrics increase as server utilization approaches 1 (100%), but they do not do so linearly or in a fixed ratio. Instead, they both exhibit an **exponential growth** pattern as utilization increases, especially nearing full capacity.

This behavior aligns with the **M/M/1 queue model**, which models a single-server system with Poisson arrivals and exponential service times. According to the theory, both **average response time** and **average queue length** grow sharply as the system utilization approaches 100%.