1 Distribution of Request Lengths

The following figure shows the distribution of request lengths handled by the server. Each bar represents the proportion of requests that fall within a specific time bin, normalized by the total number of requests.

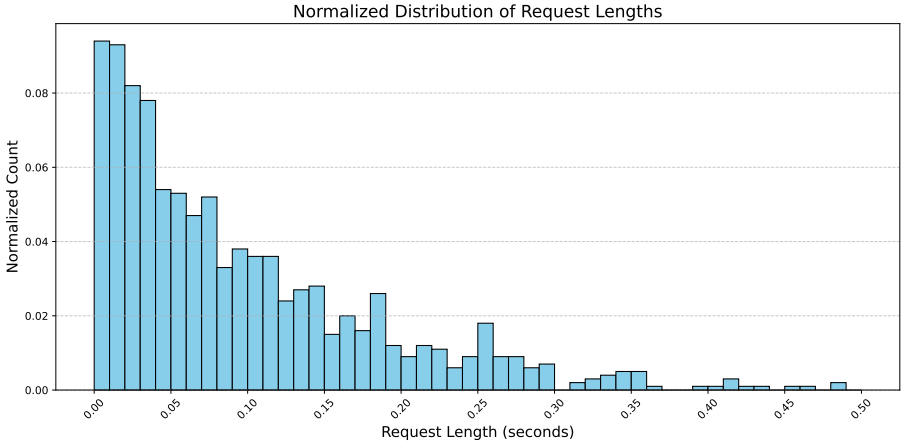

Figure 1: Enter Caption

```
pollter
import matplotlib.pyplot as plt
import numpy as np

with open('request_lengths.txt', 'r') as file:
    request_lengths = [float(line.strip()) for line in file]

bin_width = 0.005
max_length = max(request_lengths)
bins = np.arange(0, max_length + bin_width, bin_width)

counts, _ = np.histogram(request_lengths, bins=bins)

normalized_counts = counts / len(request_lengths)

bin_centers = bins[:-1] + bin_width / 2

plt.figure(figsize=(10, 6))
plt.bar(bin_centers, normalized_counts, width=bin_width, edgecolor='black')
plt.xlabel('Request Length (seconds)')
plt.ylabel('Normalized Count')
plt.title('Distribution of Request Lengths')
plt.grid(axis='y', linestyle='--', alpha=0.7)
```

```
plt.xticks(np.arange(0, max_length + bin_width, 0.005))
plt.tight_layout()
plt.show()
```

2 a distribution of the inter-arrival time

The following figure shows the distribution of inter-arrival time.

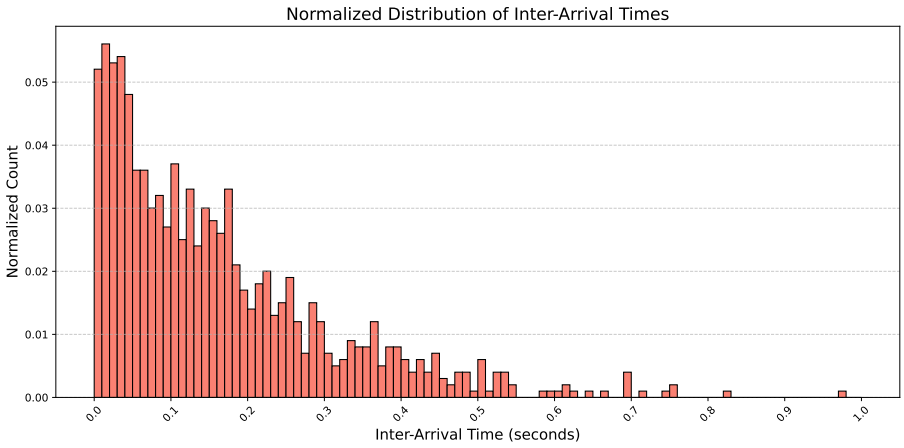

Figure 2: Enter Caption

```
import matplotlib.pyplot as plt
import numpy as np

with open('inter_arrival_times.txt', 'r') as file:
    inter_arrival_times = [float(line.strip()) for line in file]
bin_width = 0.01  # Increased from 0.005 to 0.01 seconds
max_iat = max(inter_arrival_times)
upper_limit = min(max_iat, 1.0)  # Adjust 1.0 based on your data

bins = np.arange(0, upper_limit + bin_width, bin_width)

counts, _ = np.histogram(inter_arrival_times, bins=bins)

normalized_counts = counts / len(inter_arrival_times)

bin_centers = bins[:-1] + bin_width / 2

plt.figure(figsize=(12, 6))
plt.bar(bin_centers, normalized_counts, width=bin_width, edgecolor='black', color='salmon')
plt.xlabel('Inter-Arrival Time (seconds)', fontsize=14)
```

```
plt.ylabel('Normalized Count', fontsize=14)
plt.title('Normalized Distribution of Inter-Arrival Times', fontsize=16)
plt.grid(axis='y', linestyle='--', alpha=0.7)

plt.xticks(np.arange(0, upper_limit + bin_width, 0.1), rotation=45)  # 0.1-second intervals

plt.tight_layout()
plt.savefig('inter_arrival_time_distribution_adjusted.pdf')
plt.show()
```

3 reverse-engineer distributions requestlength

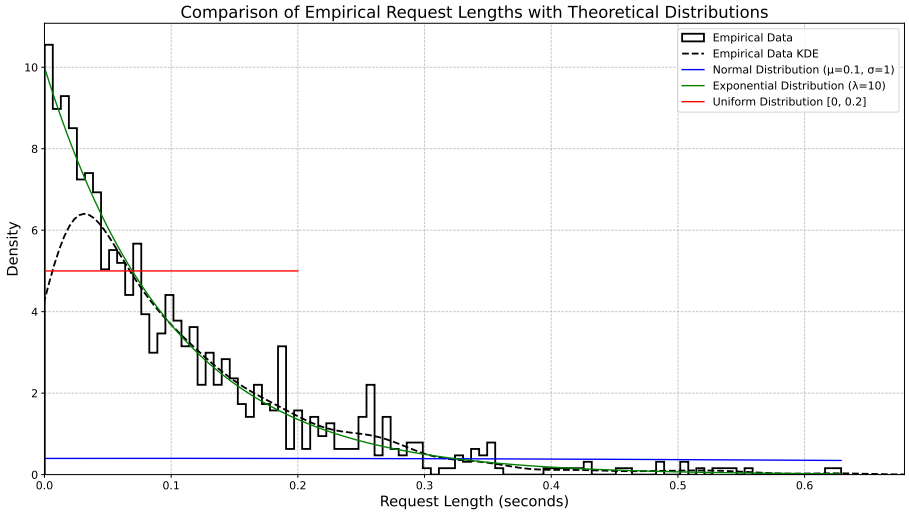

Figure 3: Enter Caption

3.1 Conclusion

Based on the overlayed distributions:

- The Exponential distribution seems to better capture the rapid decay in request lengths as they increase, aligning more closely with the empirical data's skewness.

Therefore, the Exponential distribution appears to match the empirical request length distribution more closely among the theoretical distributions considered. This suggests that request lengths may follow a memoryless process, where the probability of a request having a certain length decreases exponentially as the length increases.

```
import matplotlib.pyplot as plt
import numpy as np
from scipy.stats import norm, expon, uniform
import seaborn as sns  # For KDE plotting


empirical_data_file = 'request_lengths.txt'
with open(empirical_data_file, 'r') as file:
    empirical_request_lengths = [float(line.strip()) for line in file]

num_samples = 10000

normal_mean = 0.1
normal_std = 1
normal_samples = np.random.normal(loc=normal_mean, scale=normal_std, size=num_samples)

exponential_scale = 0.1  # scale = 1/lambda, so lambda = 10
exponential_samples = np.random.exponential(scale=exponential_scale, size=num_samples)

uniform_low = 0
uniform_high = 0.2
uniform_samples = np.random.uniform(low=uniform_low, high=uniform_high, size=num_samples)


plt.figure(figsize=(14, 8))

bins_empirical = np.linspace(min(empirical_request_lengths), max(empirical_request_lengths),
plt.hist(empirical_request_lengths, bins=bins_empirical, density=True, histtype='step', line

sns.kdeplot(empirical_request_lengths, bw_adjust=1, label='Empirical Data KDE', color='black

x_norm = np.linspace(min(empirical_request_lengths), max(empirical_request_lengths), 1000)
pdf_norm = norm.pdf(x_norm, loc=normal_mean, scale=normal_std)
plt.plot(x_norm, pdf_norm, label='Normal Distribution (=0.1, =1)', color='blue')

x_exp = np.linspace(0, max(empirical_request_lengths), 1000)
pdf_exp = expon.pdf(x_exp, scale=exponential_scale)
plt.plot(x_exp, pdf_exp, label='Exponential Distribution (=10)', color='green')

x_uni = np.linspace(uniform_low, uniform_high, 1000)
pdf_uni = uniform.pdf(x_uni, loc=uniform_low, scale=uniform_high - uniform_low)
plt.plot(x_uni, pdf_uni, label='Uniform Distribution [0, 0.2]', color='red')

plt.xlabel('Request Length (seconds)', fontsize=16)
plt.ylabel('Density', fontsize=16)
plt.title('Comparison of Empirical Request Lengths with Theoretical Distributions', fontsize
```

```
plt.legend(fontsize=12)
plt.grid(True, linestyle='--', alpha=0.7)

plt.xticks(fontsize=12)
plt.yticks(fontsize=12)

plt.xlim(0, max(empirical_request_lengths) + 0.05)

plt.tight_layout()
plt.savefig('request_length_comparison_adjusted.pdf')
plt.show()
```

4 reverse-engineer distributions interarrivaltime

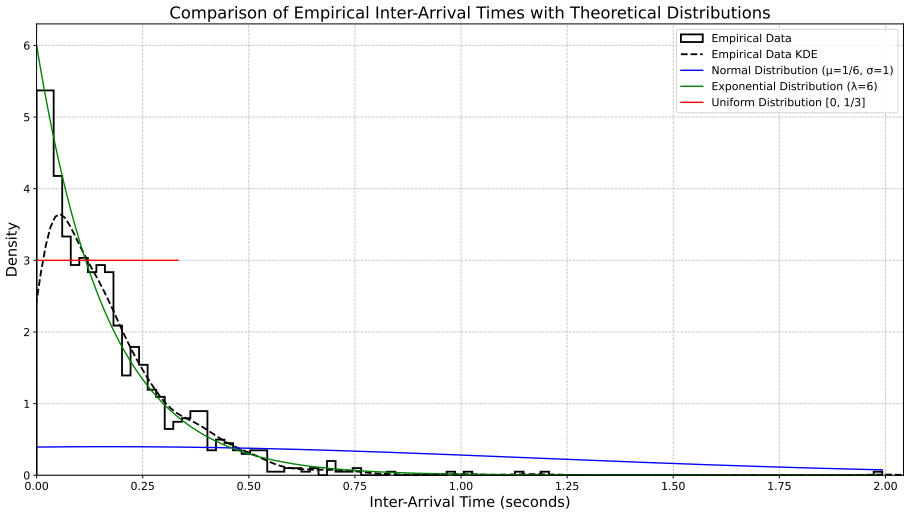

Figure 4: Enter Caption

4.0.1 Analysis

The Exponential distribution provides the closest fit to the empirical request length distribution among the theoretical distributions considered. This suggests that request lengths in the server environment may follow a memoryless process, where the probability of a request having a certain length decreases exponentially as the length increases.

4.1 Impact of Client Parameters: ‘-a‘ and ‘-s‘

- ‘-a‘ (Concurrency Level) : Controls the number of parallel request streams, directly impacting the volume and simultaneity of incoming requests.
- ‘-s‘ (Sleep Duration) : Dictates the frequency of requests sent by each thread, influencing the inter-arrival times and overall request rate.

5 queue size average

Time-Weighted Average Queue Size: 7.7987

```
        import re

# Define the log file path
LOG_FILE = 'server_log.txt'

# Regular expressions to parse the log lines
R_LINE_REGEX = re.compile(r'^R(\d+):(\d+\.\d+),')
Q_LINE_REGEX = re.compile(r'^Q:\[(.*?)\]')

def parse_log(file_path):
    snapshots = []
    current_timestamp = None

    with open(file_path, 'r') as file:
        for line in file:
            line = line.strip()

            # Check for R lines and extract timestamp
            r_match = R_LINE_REGEX.match(line)
            if r_match:
                request_id = int(r_match.group(1))
                timestamp = float(r_match.group(2))
                current_timestamp = timestamp
                continue  # Move to the next line to find the corresponding Q line

            # Check for Q lines and extract queue state
            q_match = Q_LINE_REGEX.match(line)
            if q_match and current_timestamp is not None:
                queue_content = q_match.group(1)
                if queue_content:
                    # Count the number of requests in the queue
                    queue_size = len(queue_content.split(','))
                else:
                    queue_size = 0
```

```
                # Append the snapshot as (timestamp, queue_size)
                snapshots.append((current_timestamp, queue_size))
                # Reset current_timestamp to avoid incorrect associations
                current_timestamp = None

    return snapshots

def calculate_time_weighted_average(snapshots):
    if len(snapshots) < 2:
        print("Not enough snapshots to calculate the average.")
        return None

    total_weighted = 0.0
    total_time = 0.0

    for i in range(len(snapshots) - 1):
        current_time, current_size = snapshots[i]
        next_time, _ = snapshots[i + 1]
        interval = next_time - current_time

        if interval < 0:
            print(f"Warning: Negative interval between snapshots {i} and {i+1}. Skipping.")
            continue

        total_weighted += current_size * interval
        total_time += interval

    # Handle the last snapshot's interval (assuming it lasts until the last timestamp)

    if total_time > 0:
        average = total_weighted / total_time
        return average
    else:
        print("Total time is zero. Cannot compute the average.")
        return None

def main():
    snapshots = parse_log(LOG_FILE)

    if not snapshots:
        print("No queue snapshots found in the log.")
        return

    average = calculate_time_weighted_average(snapshots)

    if average is not None:
```

```
        print(f"Time-Weighted Average Queue Size: {average:.4f}")

if __name__ == "__main__":
    main()
```

6 responed time and queue size

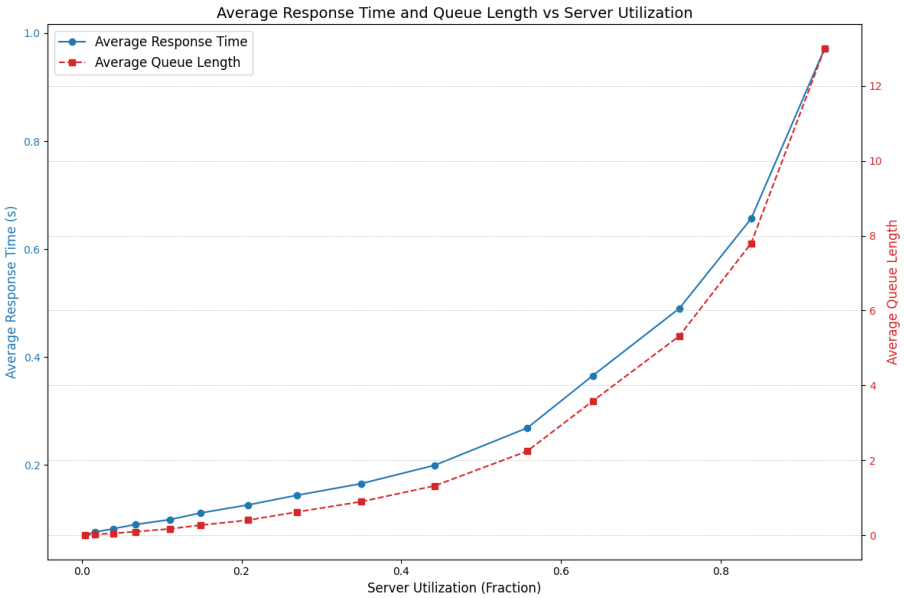

Figure 5: Enter Caption

```
        import os
import re
import pandas as pd
import sys

def compute_metrics_from_log(log_path):
    # Regular expressions to match log lines
    QUEUE_REGEX = re.compile(r'^Q:\[(.*?)\]')
    RESP_REGEX = re.compile(r'^R\d+:(\d+\.\d+),(\d+\.\d+),.*')

    queue_snapshots = []
    response_times = []

    last_r_timestamp = None
```

```
with open(log_path, 'r') as file:
    for line in file:
        line = line.strip()

        # Match R line to get timestamp and response time
        resp_match = RESP_REGEX.match(line)
        if resp_match:
            timestamp = float(resp_match.group(1))
            response_time = float(resp_match.group(2))
            response_times.append(response_time)
            last_r_timestamp = timestamp
            continue

        # Match queue snapshot
        queue_match = QUEUE_REGEX.match(line)
        if queue_match and last_r_timestamp is not None:
            queue_content = queue_match.group(1)
            if queue_content:
                queue_size = len(queue_content.split(','))
            else:
                queue_size = 0
            queue_snapshots.append((last_r_timestamp, queue_size))
            # Reset last_r_timestamp to avoid associating multiple Q lines with one R li
            last_r_timestamp = None
            continue

# Compute Utilization
if len(queue_snapshots) < 2:
    utilization = 0.0
    average_queue_length = 0.0
else:
    # Sort snapshots by timestamp
    queue_snapshots.sort(key=lambda x: x[0])

    total_time = 0.0
    busy_time = 0.0
    weighted_sum = 0.0

    for i in range(len(queue_snapshots) - 1):
        current_time, current_size = queue_snapshots[i]
        next_time, _ = queue_snapshots[i + 1]
        interval = next_time - current_time

        if interval < 0:
            continue  # Skip invalid intervals
```

```
            total_time += interval
            if current_size > 0:
                busy_time += interval

            weighted_sum += current_size * interval

        if total_time > 0:
            utilization = busy_time / total_time
            average_queue_length = weighted_sum / total_time
        else:
            utilization = 0.0
            average_queue_length = 0.0

    # Compute Average Response Time
    if response_times:
        average_response_time = sum(response_times) / len(response_times)
    else:
        average_response_time = 0.0

    return utilization, average_response_time, average_queue_length

def main():
    # Define the directory containing experiment logs
    LOG_DIR = 'experiment_logs'

    if not os.path.isdir(LOG_DIR):
        print(f"Error: Directory '{LOG_DIR}' does not exist.")
        sys.exit(1)

    # Prepare a list to store metrics for each experiment
    metrics_list = []

    # Iterate through each experiment log
    for filename in os.listdir(LOG_DIR):
        if filename.startswith('experiment_a') and filename.endswith('.log'):
            a_value_matches = re.findall(r'experiment_a(\d+)\.log', filename)
            if not a_value_matches:
                print(f"Warning: Filename '{filename}' does not match expected pattern.")
                continue
            a_value = int(a_value_matches[0])
            log_path = os.path.join(LOG_DIR, filename)

            print(f"Processing {filename}...")

            try:
```

```
                utilization, average_response_time, average_queue_length = compute_metrics_f
                print(f"Metrics for a={a_value}: Utilization={utilization:.4f}, "
                      f"Avg Response Time={average_response_time:.6f}, "
                      f"Avg Queue Length={average_queue_length:.4f}")
            except Exception as e:
                print(f"Error processing {filename}: {e}")
                continue

            metrics_list.append({
                'a': a_value,
                'utilization': utilization,
                'average_response_time': average_response_time,
                'average_queue_length': average_queue_length
            })

    if not metrics_list:
        print("No metrics to aggregate. Please check your log files.")
        sys.exit(1)

    # Create a DataFrame from the metrics list
    metrics_df = pd.DataFrame(metrics_list)

    # Sort the DataFrame by 'a'
    metrics_df.sort_values(by='a', inplace=True)

    # Save the aggregated data to a CSV file
    metrics_df.to_csv('aggregated_results.csv', index=False)

    print("Aggregated metrics saved to 'aggregated_results.csv'.")

if __name__ == "__main__":
    main()
```

6.1

Relationship Between Response Time and Queue Length with Increasing Utilization

As **server utilization** increases, both the **average response time** and **average queue length** exhibit a strong, positive correlation, evolving in accordance with queuing theory principles. Specifically, the data shows the following trends:

- The **average response time** increases from **0.0704 seconds** at 0.38% utilization to **0.9714 seconds** at 92.96% utilization.

- The **average queue length** similarly increases from **0.0041** to **12.9944** over the same range of utilization.

This indicates that **higher server utilization** results in both **longer queues** and **slower response times**, with both metrics escalating significantly as utilization rises.

6.2 Conclusion

Therefore, as **server utilization increases**, the relationship between **average response time** and **average queue length** is a strong positive correlation, with both metrics escalating due to the system nearing saturation.

7

modeling relationship The Pearson Correlation Coefficient between Average Response Time (W) and Average Queue Length (L_q) is calculated to be approximately $r = 0.9998$, with a p-value of $p < 0.0001$. This near-perfect correlation suggests that as one metric increases, the other does so in a nearly linear fashion.