Experiment Report

Experiment 1: Prime Number Computation Endpoint

Aim/purpose of experiment

The experiment aims to evaluate the performance of the web server implemented using libhttp library by simulating concurrent requests and measuring throughput on /arithmetic/prime endpoint.

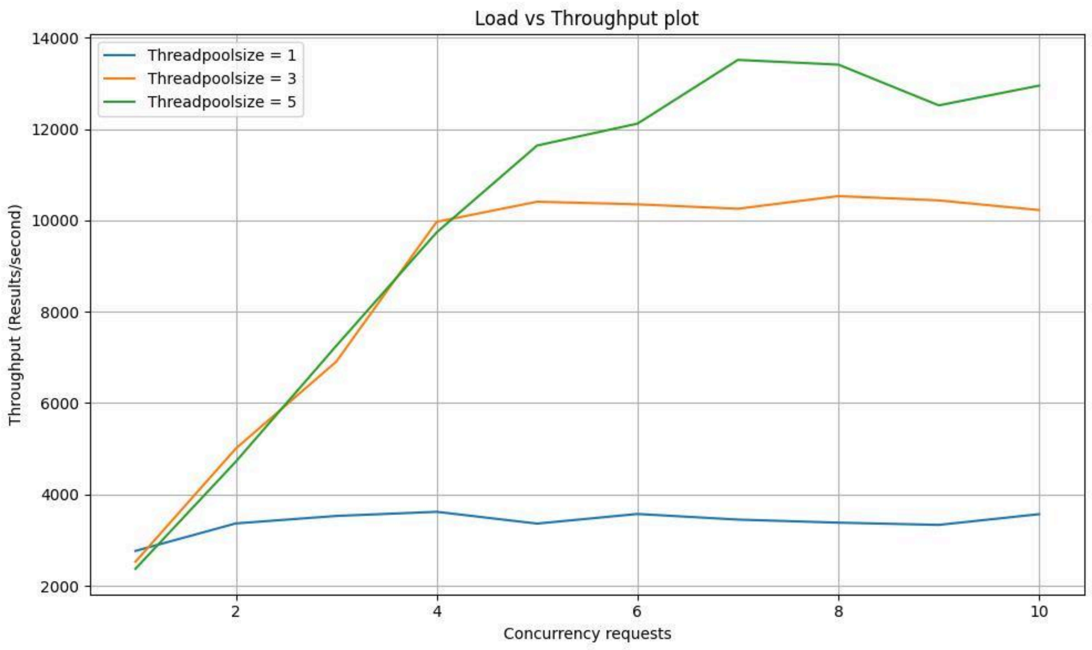
Setup and Execution Details

- Load Generator: Apache Benchmark (ab)
- Server Configuration
  - PORT: 9999
  - Thread Pool Size: varied as 1,3 and 5
- Independent Parameter: Number of concurrent requests (varied as 1,2,3,4,5,6,7,8,9,10)
- Execution Command: ab -n 5000 -c [concurrent_requests] http://localhost:9999/arithmetic/prime?num=999999937
- Metrics recorded
  - Requests per second (throughput)

Hypothesis/Expectation

With the increasing number of concurrent requests, we expect throughput to initially rise as more threads are used, then plateau or decrease eventually due to the server's resource limitations.

Observations from the Data/Plots



- Thread pool size = 1: Throughput remains almost stable between 2700 req/sec and 3600 req/sec across varying levels of concurrency.
- Thread pool size = 3: Throughput generally increases up to 4 concurrent requests, reaching a maximum of 10530 req/sec before plateauing.
- Thread pool size = 5: Throughput generally increases up to 7 concurrent requests, reaching a maximum of 13512 req/sec before plateauing.

Also when thread pool size increases the throughput increases as seen from the experiment.

Explanation of Behavior and Inferences

As we expected throughput first increases as the number of concurrent requests increases and then plateaued at higher concurrency levels.

This behavior confirms that the server can handle a limited number of concurrent requests effectively before performance begins to degrade due to limited resources or thread management overhead.

Experiment 2: Ping-Pong String Echo Endpoint

Aim/purpose of experiment

The experiment aims to evaluate the performance of the web server implemented using libhttp library by simulating concurrent requests and measuring throughput on /pingpong endpoint.

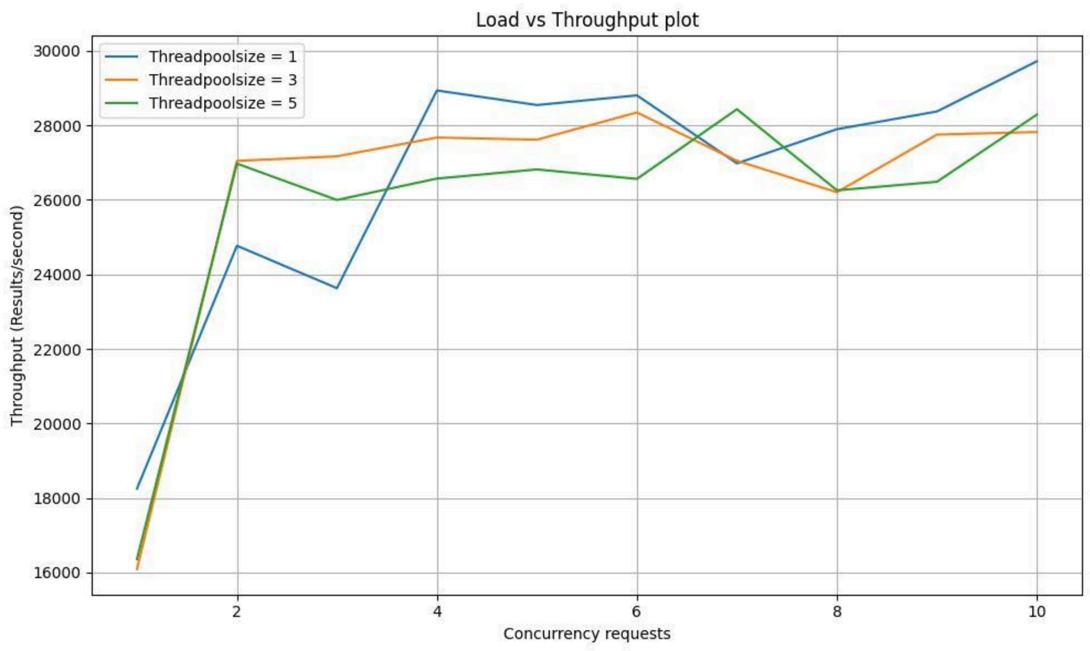
Setup and Execution Details

- Load Generator: Apache Benchmark (ab)
- Server Configuration
  - PORT: 9999
  - Thread Pool Size: varied as 1,3 and 5
- Independent Parameter: Number of concurrent requests (varied as 1,2,3,4,5,6,7,8,9,10)
- Execution Command: ab -n 5000 -c [concurrent_requests] http://localhost:9999/pingpong?str=Nishant
- Metrics recorded
  - Requests per second (throughput)

Hypothesis/Expectation

When the number of concurrent requests increases, we expect that throughput will first increase, then plateau or decrease due to server resource limitations.

Observations from the Data/Plots



- Thread pool size = 1: Throughput generally increases up to 4 concurrent requests, reaching a maximum of 28937 req/sec before plateauing.
- Thread pool size = 3: Throughput generally increases up to 2 concurrent requests, reaching a maximum of 27169 req/sec before plateauing.
- Thread pool size = 5: Throughput generally increases up to 2 concurrent requests, reaching a maximum of 26973 req/sec before plateauing.

Also when the thread pool size increases the throughput remains the same across each thread pool size. This is because the server's resources for each request are not that much used as we are just returning the string. We also see the spikes in throughput after plateauing because of system dynamics like network congestion, latency, etc.

Explanation of Behavior and Inferences

As we expected the throughput first increased as the number of concurrent requests increased and then plateaued at higher concurrency levels. Given the simplicity of /pingpong endpoint, the server achieves maximum throughput without heavy resource demands, and hence thread pool size adjustments have less impact on performance.

This result confirms that the server can only handle a limited number of concurrent requests effectively, and then after that performance starts to degrade due to the limited number of resources or thread management issues.

Experiment 3: Fibonacci Computation Endpoint

Aim/purpose of experiment

The experiment aims to evaluate the performance of the web server implemented using libhttp library by simulating concurrent requests and measuring throughput on /arithmetic/fibonacci endpoint.

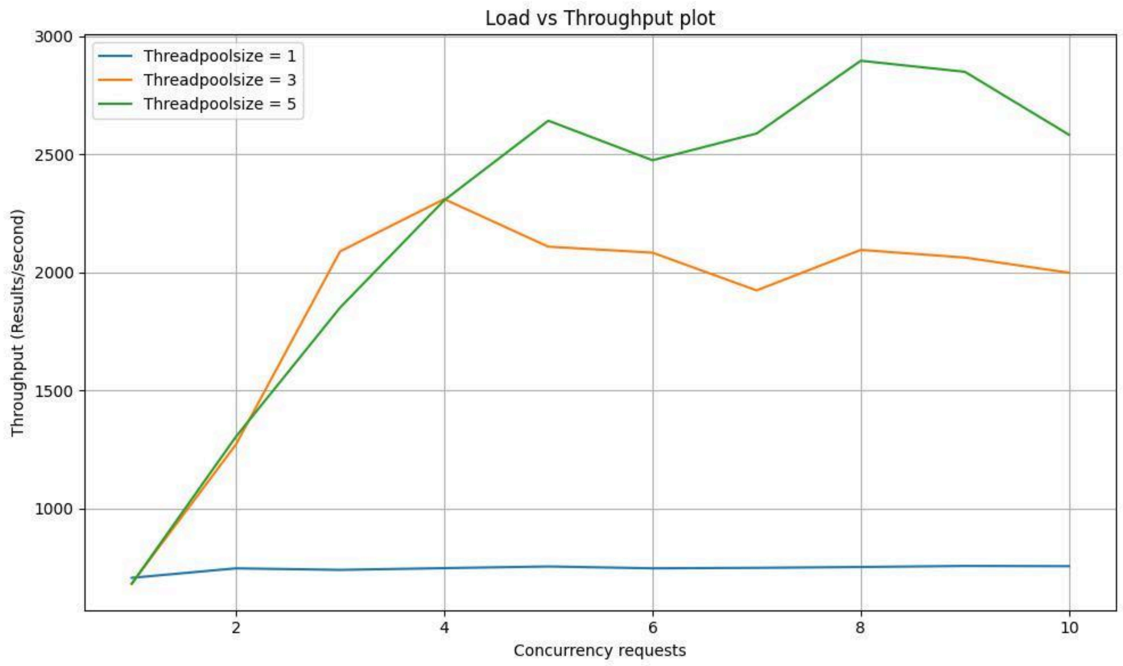
Setup and Execution Details

- Load Generator: Apache Benchmark (ab)
- Server Configuration
    - PORT: 9999
    - Thread Pool Size: varied as 1,3 and 5
- Independent Parameter: Number of concurrent requests (varied as 1,2,3,4,5,6,7,8,9,10)
- Execution Command: ab -n 5000 -c [concurrent_requests] http://localhost:9999/arithmetic/fibonacci?num=500000
- Metrics recorded
    - Requests per second (throughput)

Hypothesis/Expectation

When the number of concurrent requests increases, we expect that throughput will first increase, then plateau or decrease due to server resource limitations.

Observations from the Data/Plots



- Thread pool size = 1: Throughput stabilizes between 707req/sec and 757 req/sec with minimal variations.
- Thread pool size = 3: Throughput generally increases up to 4 concurrent requests, reaching a maximum of 2310 req/sec before plateauing.
- Thread pool size = 5: Throughput generally increases up to 5 concurrent requests, reaching a maximum of 2641 req/sec before plateauing. (It spikes highest at the 8th concurrent level, reaching 2895 req/sec, this varies on system dynamics).

Also when thread pool size increases the throughput increases as seen from the experiment.

Explanation of Behavior and Inferences

As we expected the throughput first increased as the number of concurrent requests increased and then plateaued at higher concurrency levels. This is because the server's resource limitation is becoming a bottleneck.

Hence, this result confirms that the server can only handle a limited number of concurrent requests effectively, and then after that performance starts to degrade due to the limited number of resources or thread management issues.