Assignment 1 Report

Github Repo:

https://github.com/peihsuan-lin/cs6650/tree/main/assignment1

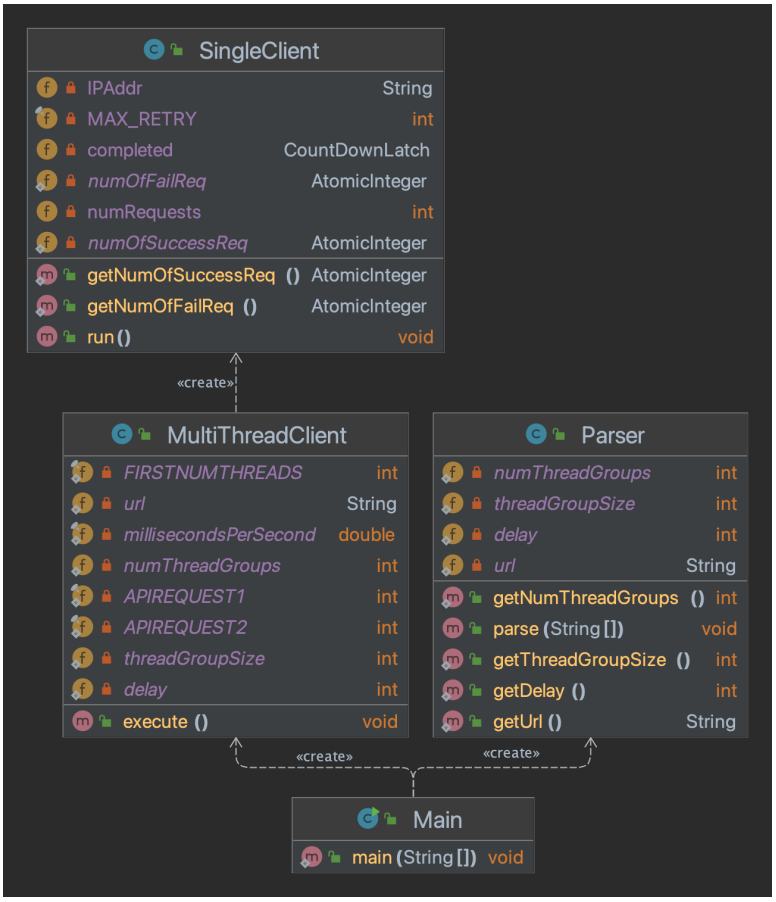
Client Design

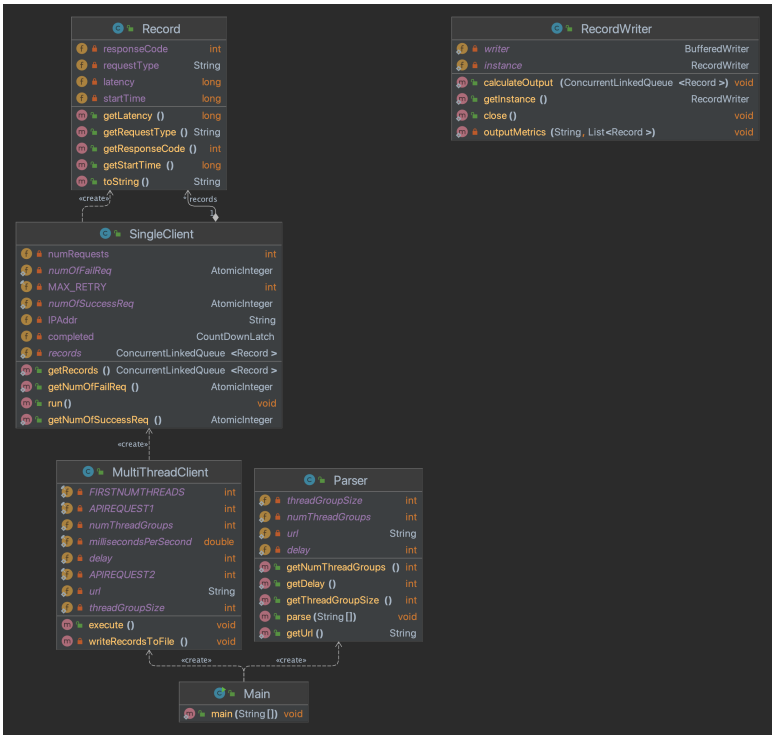
Packages and Major Classes

The client application is organized into two separate packages named `part1` and `part2`. Both parts share similar structures but have some unique features. Here are the major classes:

1. **Main.java**: Entry point for the client application in both parts. Initializes the parser and the multi-threaded client.
2. **Parser.java**: Parses command-line arguments and configures the client behavior in both parts.
3. **SingleClient.java**: Represents a single client that performs HTTP requests in both parts.
4. **MultiThreadClient.java**: Manages multiple threads for making HTTP requests in both parts.
5. **Record.java**: Unique to part 2, this class stores information about each HTTP request.
6. **RecordWriter.java**: Unique to part 2, this class writes the records to a file.

Class Relationships and Responsibilities


SingleClient
IPAddr String
MAX_RETRY int
completed CountDownLatch
numOfFailReq AtomicInteger
numRequests int
numOfSuccessReq AtomicInteger
getNumOfSuccessReq () AtomicInteger
getNumOfFailReq () AtomicInteger
run () void
«create»
MultiThreadClient
FIRSTNUMTHREADS int
url String
millisecondsPerSecond double
numThreadGroups int
APIREQUEST1 int
APIREQUEST2 int
threadGroupSize int
delay int
execute () void
Parser
numThreadGroups int
threadGroupSize int
delay int
url String
getNumThreadGroups () int
parse (String []) void
getThreadGroupSize () int
getDelay () int
getUrl () String
«create»
«create»
Main
main (String []) void


Record
responseCode int
requestType String
latency long
startTime long
getLatency () long
getRequestType () String
getResponseCode () int
getStartTime () long
toString () String
«create»
records
RecordWriter
writer BufferedWriter
instance RecordWriter
calculateOutput (ConcurrentLinkedQueue <Record >) void
getInstance () RecordWriter
close () void
outputMetrics (String, List<Record >) void
SingleClient
numRequests int
numOfFailReq AtomicInteger
MAX_RETRY int
numOfSuccessReq AtomicInteger
IPAddr String
completed CountDownLatch
records ConcurrentLinkedQueue <Record >
getRecords () ConcurrentLinkedQueue <Record >
getNumOfFailReq () AtomicInteger
run () void
getNumOfSuccessReq () AtomicInteger
«create»
MultiThreadClient
FIRSTNUMTHREADS int
APIREQUEST1 int
numThreadGroups int
millisecondsPerSecond double
delay int
APIREQUEST2 int
url String
threadGroupSize int
execute () void
writeRecordsToFile () void
Parser
threadGroupSize int
numThreadGroups int
url String
delay int
getNumThreadGroups () int
getDelay () int
getThreadGroupSize () int
parse (String []) void
getUrl () String
«create»
«create»
Main
main (String []) void

1. Main.java

Responsibilities: Initializes the `Parser` and `MultiThreadClient` classes and triggers the execution.

2. Parser.java

Responsibilities: Parses command-line arguments to configure the client.

3. SingleClient and MultiThreadClient

By separating the `SingleClient` and `MultiThreadClient` classes, the design adheres to the SOLID principles. The `SingleClient` class is responsible for the behavior of an individual client, such as making HTTP requests and handling responses. The `MultiThreadClient` class is responsible for managing multiple threads, each represented by a `SingleClient` instance.

- **SingleClient.java**
  - **Responsibilities**: Performs HTTP requests and keeps track of successful and failed requests.
  - **Unique to Part 2**: Also stores request information in `Record` objects and adds them to a queue.
- **MultiThreadClient.java**
  - **Responsibilities**: Manages the execution of multiple threads for making HTTP requests.
  - **Unique to Part 2**: May include additional error handling and logging.

4. Record.java (Part 2)

Responsibilities: Stores information about an HTTP request, such as the start time, request type, latency, and response code.

5. RecordWriter.java (Part 2)

Responsibilities: Writes the records of HTTP requests to a file. Follows the Singleton pattern.

Execution Flow

1. The `Main` class initializes the `Parser` and passes the command-line arguments for parsing.
2. The `Parser` class configures the client by setting various attributes.
3. `Main` then initializes the `MultiThreadClient` class and calls its `execute()` method.
4. `MultiThreadClient` launches multiple `SingleClient` threads according to different test loads for making HTTP requests.

Threading and Synchronization

By combining these mechanisms, the client application performs multi-threaded HTTP requests. Part 2 adds the ability to spot performance of each request.

Threading Model

The client application uses a multi-threaded approach to perform HTTP requests concurrently. It employs Java's native threading capabilities, where each thread is represented by an instance of the `SingleClient` class.

CountDownLatch

The application uses Java's `CountDownLatch` class to ensure that all threads have completed their execution before the program terminates. This is for synchronizing the start and end points of multiple threads. The latch is initialized with the total number of threads and is decremented as each thread completes its task. Once the count reaches zero, the latch triggers, indicating that all threads have finished.

AtomicInteger

To keep track of successful and failed HTTP requests across multiple threads, the application uses `AtomicInteger` variables. These provide a thread-safe way to increment or decrement a counter used for collecting success/failure requests. Since multiple threads might try to update the same counter, `AtomicInteger` ensures that the operation is atomic and prevents race conditions.

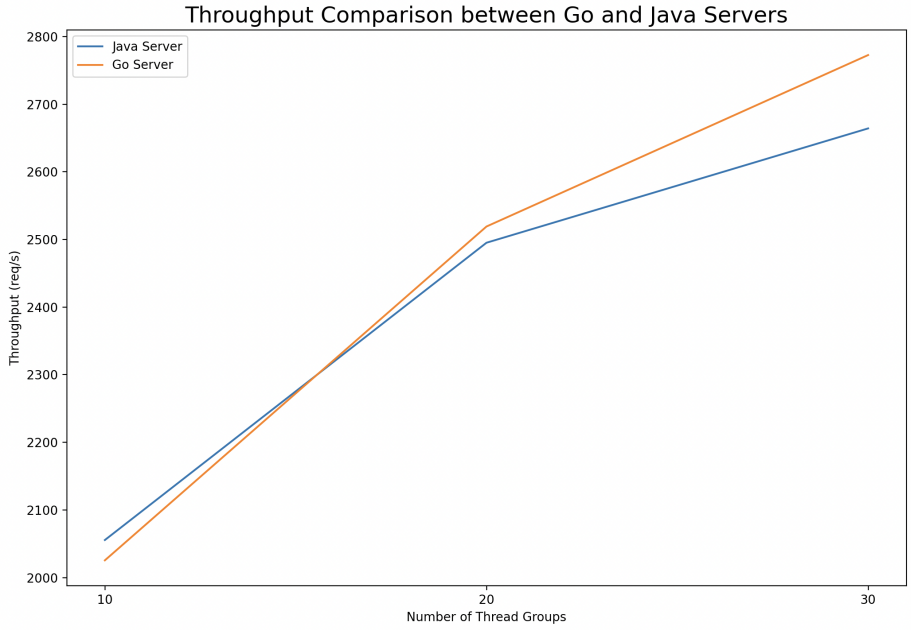
Record Queue

Unique to Part 2, the application uses a `ConcurrentLinkedQueue` to store `Record` objects that contain information about each HTTP request. This data structure is thread-safe and allows multiple threads to add records simultaneously.

Singleton Pattern in RecordWriter

In Part 2, the `RecordWriter` class follows the Singleton pattern to ensure that there is only one instance of the writer. This avoids potential issues related to file writing when multiple threads are involved. The writer is responsible for writing the records to a file in a synchronized manner.

Client (Part 1)



Test Conditions

Three different test loads were run for both Go and Java servers. The following parameters were used for the test loads is based on `threadGroupSize = 10, delay = 2`.

The number of thread groups varied among 10, 20, 30.

Server Output

GO

Java

```
Go Server
Test load:
threadGroupSize: 10, numThreadGroups: 10, delay: 2
Time taken: 49866 ms
Number of successful requests: 101000
Number of fail requests: 0
Walltime: 49.864 seconds
Total throughput: 2025.509385528638 req/s
```

```
Java Server
Test load:
threadGroupSize: 10, numThreadGroups: 10, delay: 2
Time taken: 49138 ms
Number of successful requests: 101000
Number of fail requests: 0
Walltime: 49.136 seconds
Total throughput: 2055.519374796483 req/s
```

```
Go Server
Test load:
threadGroupSize: 10, numThreadGroups: 20, delay: 2
Time taken: 79793 ms
Number of successful requests: 201000
Number of fail requests: 0
Walltime: 79.791 seconds
Total throughput: 2519.0810993721097 req/s
```

```
Java Server
Test load:
threadGroupSize: 10, numThreadGroups: 20, delay: 2
Time taken: 80560 ms
Number of successful requests: 201000
Number of fail requests: 0
Walltime: 80.558 seconds
Total throughput: 2495.096700513915 req/s
```

```
Go Server
Test load:
threadGroupSize: 10, numThreadGroups: 30, delay: 2
Time taken: 108565 ms
Number of successful requests: 301000
Number of fail requests: 0
Walltime: 108.563 seconds
Total throughput: 2772.5836611000063 req/s
```

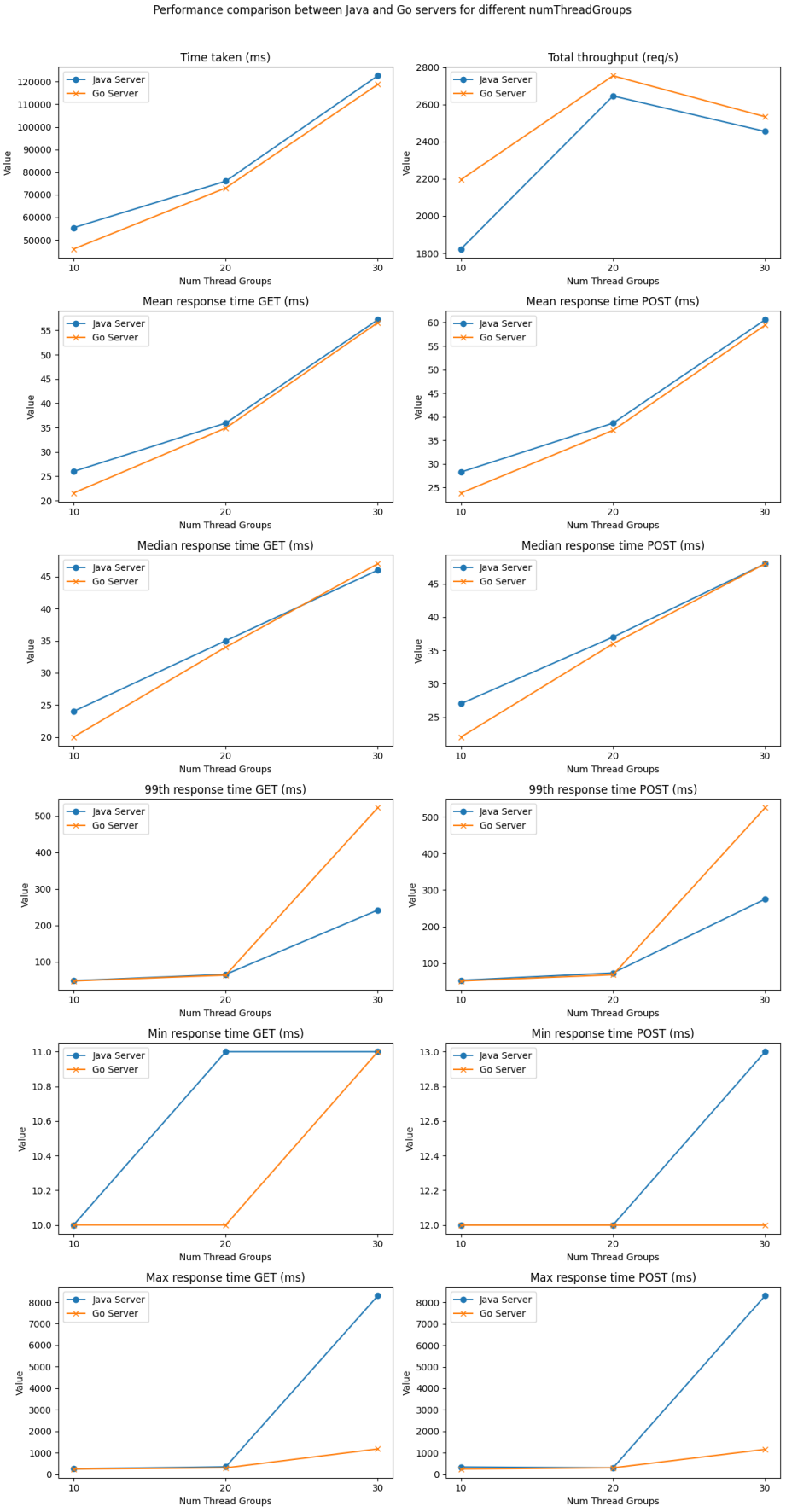
```
Java Server
Test load:
threadGroupSize: 10, numThreadGroups: 30, delay: 2
Time taken: 112985 ms
Number of successful requests: 301000
Number of fail requests: 0
Walltime: 112.983 seconds
Total throughput: 2664.117610613986 req/s
```

Observations

1. **Increasing Throughput with More Thread Groups**: Both servers display an upward trend in throughput as the number of thread groups increases. This indicates that both systems scale positively with an increase in load.
2. **Higher Throughput for Go Server**: Across all tested scenarios, the Go Server consistently outperforms the Java Server in terms of throughput. The gap widens further when the number of thread groups is increased to 30, suggesting better scalability for the Go Server.

Client (Part 2)

Overall Statistics


Performance comparison between Java and Go servers for different numThreadGroups
Time taken (ms)
Total throughput (req/s)
Mean response time GET (ms)
Mean response time POST (ms)
Median response time GET (ms)
Median response time POST (ms)
99th response time GET (ms)
99th response time POST (ms)
Min response time GET (ms)
Min response time POST (ms)
Max response time GET (ms)
Max response time POST (ms)
Java Server
Go Server
Value
Num Thread Groups

Server Output

`numThreadGroup=10`

GO

```
Test load:
threadGroupSize: 10, numThreadGroups: 10, delay: 2
Time taken: 46001 ms
Number of successful requests: 101000
Number of fail requests: 0
walltime: 45.999 seconds
Total throughput: 2195.699906519707 req/s

Metrics for GET:
Mean response time: 21.54753465346535 ms
Median response time: 20.0 ms
99th response time: 48.0 ms
Min response time: 10.0 ms
Max response time: 246.0 ms

Metrics for POST:
Mean response time: 23.83568316831683 ms
Median response time: 22.0 ms
99th response time: 51.0 ms
Min response time: 12.0 ms
Max response time: 255.0 ms
```

Java

```
Test load:
threadGroupSize: 10, numThreadGroups: 10, delay: 2
Time taken: 55416 ms
Number of successful requests: 101000
Number of fail requests: 0
walltime: 55.414 seconds
Total throughput: 1822.6440971595625 req/s

Metrics for GET:
Mean response time: 25.977821782178218 ms
Median response time: 24.0 ms
99th response time: 49.0 ms
Min response time: 10.0 ms
Max response time: 256.0 ms

Metrics for POST:
Mean response time: 28.27487128712871 ms
Median response time: 27.0 ms
99th response time: 53.0 ms
Min response time: 12.0 ms
Max response time: 349.0 ms
```

`numThreadGroup=20`

GO

```
Test load:
threadGroupSize: 10, numThreadGroups: 20, delay: 2
Time taken: 72978 ms
Number of successful requests: 201000
Number of fail requests: 0
walltime: 72.976 seconds
Total throughput: 2754.330190747643 req/s

Metrics for GET:
Mean response time: 34.89301492537314 ms
Median response time: 34.0 ms
99th response time: 64.0 ms
Min response time: 10.0 ms
Max response time: 299.0 ms

Metrics for POST:
Mean response time: 37.11939303482587 ms
Median response time: 36.0 ms
99th response time: 68.0 ms
Min response time: 12.0 ms
Max response time: 309.0 ms
```

Java

```
Test load:
threadGroupSize: 10, numThreadGroups: 20, delay: 2
Time taken: 75979 ms
Number of successful requests: 201000
Number of fail requests: 0
walltime: 75.977 seconds
Total throughput: 2645.5374652855467 req/s

Metrics for GET:
Mean response time: 35.91497512437811 ms
Median response time: 35.0 ms
99th response time: 66.0 ms
Min response time: 11.0 ms
Max response time: 347.0 ms

Metrics for POST:
Mean response time: 38.637149253731344 ms
Median response time: 37.0 ms
99th response time: 73.0 ms
Min response time: 12.0 ms
Max response time: 309.0 ms
```

`numThreadGroup=30`

GO

```
Test load:
threadGroupSize: 10, numThreadGroups: 30, delay: 2
Time taken: 118795 ms
Number of successful requests: 301000
Number of fail requests: 0
walltime: 118.793 seconds
Total throughput: 2533.819332788969 req/s

Metrics for GET:
Mean response time: 56.59259468438538 ms
Median response time: 47.0 ms
99th response time: 522.0 ms
Min response time: 11.0 ms
Max response time: 1180.0 ms

Metrics for POST:
Mean response time: 59.431714285714285 ms
Median response time: 48.0 ms
99th response time: 525.0 ms
Min response time: 12.0 ms
Max response time: 1168.0 ms
```

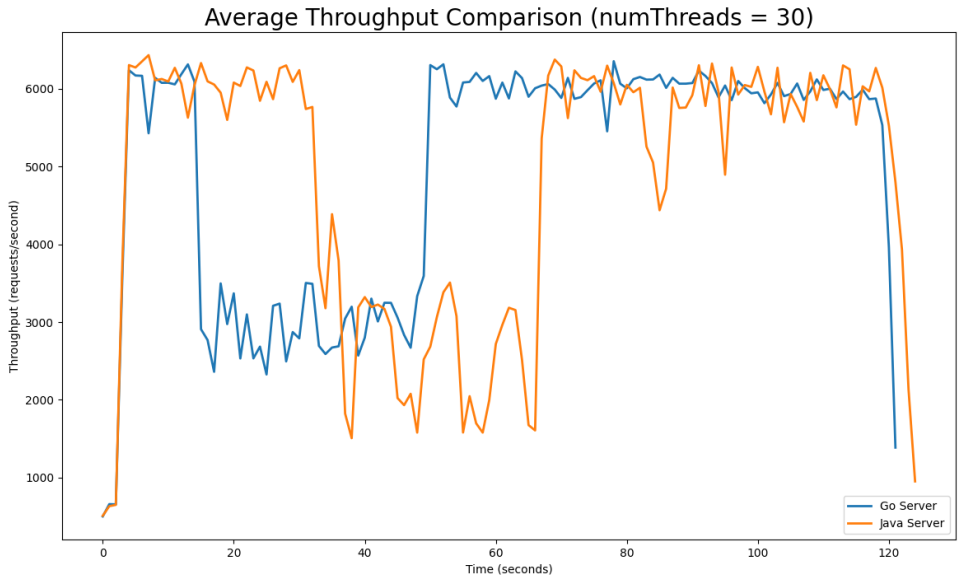
Java

```
Test load:
threadGroupSize: 10, numThreadGroups: 30, delay: 2
Time taken: 122615 ms
Number of successful requests: 301000
Number of fail requests: 0
walltime: 122.613 seconds
Total throughput: 2454.8783571073213 req/s

Metrics for GET:
Mean response time: 57.179109634551494 ms
Median response time: 46.0 ms
99th response time: 242.0 ms
Min response time: 11.0 ms
Max response time: 8297.0 ms

Metrics for POST:
Mean response time: 60.570394413363275 ms
Median response time: 48.0 ms
99th response time: 275.0 ms
Min response time: 13.0 ms
Max response time: 8297.0 ms
```

Performance Comparison for a single test



Overview

This document provides an analysis of the performance comparison between a Go Server and a Java Server based on test configuration:

- `threadGroupSize = 10`
- `numThreadGroups = 30`
- `delay = 2`

The primary focus is on throughput (requests per second) and various response time metrics.

Chart Analysis

Description

The chart visualizes the throughput performance of both servers over time.

- **X-axis**: Time in seconds, starting at 0 and extending to the maximum test duration for each server.
- **Y-axis**: Throughput in requests per second.

Observations

	Java Server	**Go Server**
Throughput	2454.88 req/s	2533.82 req/s
Walltime	122.613 seconds	118.793 seconds
Response Time		
Mean	57.18 ms (GET), 60.57 ms (POST)	56.59 ms (GET), 59.43 ms (POST)
Median	46 ms (GET), 48 ms (POST)	47 ms (GET), 48 ms (POST)
99th Percentile	242 ms (GET), 275 ms (POST)	522 ms (GET), 525 ms (POST)
Max	8297.0 ms (GET), 8297.0 ms (POST)	1180.0 ms (GET), 1168.0 ms (POST)
Min	11.0 ms (GET), 13.0 ms (POST)	11.0 ms (GET), 12.0 ms (POST)

- The Go Server has a slightly lower mean and median response time. However, its 99th percentile response time is significantly higher, indicating potential performance issues under extreme conditions.

- The Go Server shows a slightly better performance in terms of throughput and average response time. However, its higher 99th percentile response time indicates that there may be performance bottlenecks or stability issues under extreme conditions.