






# Term Project: Final Report

## Multithreaded Web Server

### Operating Systems (EE-463)

Name	ID	Signature
Ahmed Abdulaziz Alzbidi	1651491	
Khalid Waleed Saqi	1741869	
Mohammed Abdullah Alsaggaf	1740489	

Date: 20<sup>th</sup>, April 2021

Instructor: Dr. Abdulghani Alqasimi



## Contents

<b>Chapter 1 Design Documentation</b> .....	<b>2</b>
<b>1.1. Introduction</b> .....	<b>2</b>
<b>1.2. Problem Definition</b> .....	<b>2</b>
<b>1.3. Design Block Diagram</b> .....	<b>2</b>
<b>1.4. Design Specifications</b> .....	<b>4</b>
1.4.1 Main thread.....	4
1.4.2 Monitor thread .....	5
1.4.3 Thread Pool .....	7
1.4.4 Worker Thread .....	7
1.4.5 Circler Queue data structure .....	8
1.4.6 Request class .....	9
<b>1.5. Project Schedule</b> .....	<b>10</b>
<b>1.6. Team Schedule</b> .....	<b>11</b>
<b>1.7. Conclusion</b> .....	<b>11</b>
<b>Chapter 2 Benchmark Test</b> .....	<b>12</b>
<b>2.1. Introduction</b> .....	<b>12</b>
<b>2.2. Objectives</b> .....	<b>12</b>
<b>2.3. Device Specifications</b> .....	<b>12</b>
<b>2.4. Results</b> .....	<b>12</b>
2.4.1. Optimal Error-free Settings .....	12
2.4.2. Optimal Error-free Settings Analysis .....	17
2.4.3. Overload Policies Performance .....	19
2.4.3.1 DPRT.....	19
2.4.3.2 DPRH .....	21
<b>2.5. Suggestions</b> .....	<b>23</b>

# Chapter 1 Design Documentation

## 1.1. Introduction

In this project we assigned to work on a web server to practice what we learned in the theoretical lectures such as, message-passing in a client-server setup, shared memory and signals, multi-threading, synchronization, and mutual exclusion techniques. We provided a single threaded web server, and our objective is to convert it to a multithreaded web server. So, we will do some changes on the main thread. Also, we will add a monitor thread, thread pool and circular queue. Also, we will handle some overload situations. These parts will explain very well in next points.

## 1.2. Problem Definition

The provided web server is very limited. It is serving one user request at time because the web server's architecture is single-threaded and that is workable but not efficient.

## 1.3. Design Block Diagram

The server design is following the multithreading architecture to enable the concurrent handling of the web requests. Basically, the design consists of four main components, the main thread, the monitor thread, the CircularQueue data structure and the ThreadPool data structure. At the beginning the main thread will initialize all the required components including the monitor thread which then will initialize the ThreadPool with the workers number specified by the user. The main thread will receive the HTTP requests and write them into the CircularQueue and the workers thread will read from the queue to serve these requests. The monitor thread will monitor the ThreadPool and handle any dead workers in addition to handle a termination signal (Ctrl-c) and any abnormal conditions and report it to the web-server-log (stdout). The threads responsibilities can be summarized as follow:

### 1- Main Thread

- a. Generate Monitor Thread.
- b. Setup Socket connection.
- c. Accept Web requests.
- d. Build The request structure and write to the CircularQueue object.
- e. Handle the overload conditions.

### 2- Monitor Thread

- a. Initialize ThreadPool object.
- b. Maintain the number of the live workers inside the ThreadPool.
- c. Report any abnormal conditions to the stdout.
- d. Handle a termination signal (Ctrl-c) when it comes from the keyboard.
- e. Clean-up before normal termination.

To facilitate these responsibilities to the threads. The created data structures must provide the following:

**1- Class CircularQueue**

- a. Protected writing to the queue.
- b. Protected FIFO reading from the queue.
- c. Protected removing from the queue.
- d. Capacity checking.

**2- Class ThreadPool**

- a. Initialize and start the workers thread.
- b. Maintain number of live workers threads.
- c. Clean up all the worker threads.
- d. Check workers thread conditions.

**3- Class Worker**

- a. Inherit class Thread utilities.
- b. Read and remove Request from Queue.
- c. Serve Request.
- d. Delete Request object.

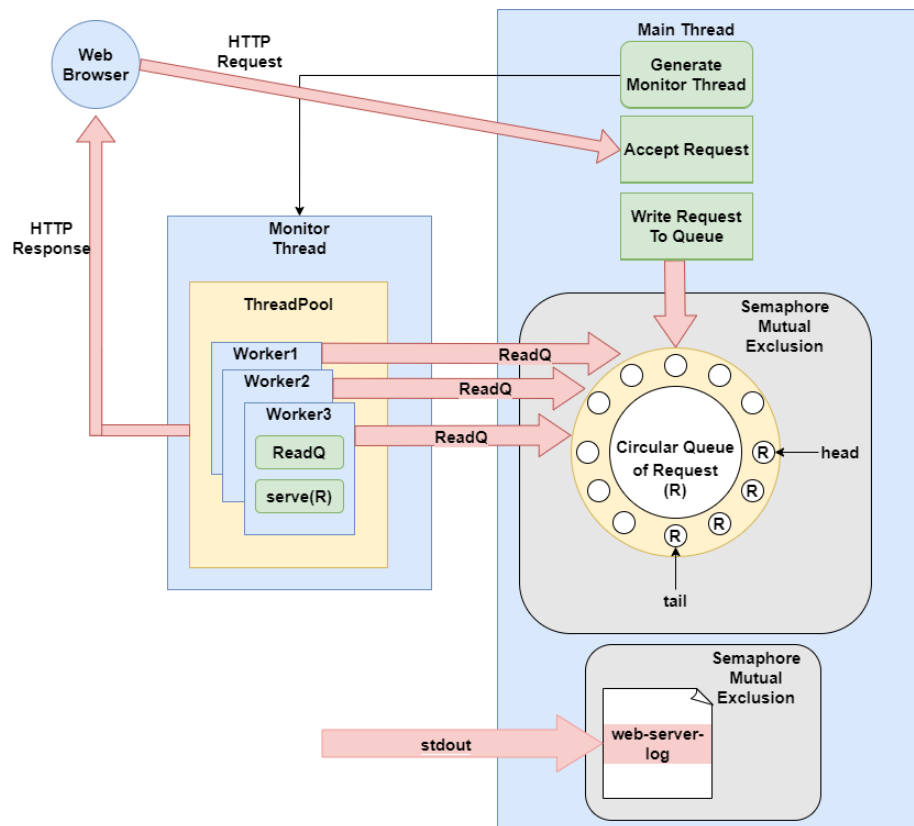


Figure 1 Block diagram

## 1.4. Design Specifications

This section will illustrate clearly all the requirements for all the design components. The illustrated parts are Main thread, Monitor thread, Pool data structure, Worker Data structure and Circular Queue data structure. Reasons for choosing these designs will be provided for each section.

### 1.4.1 Main thread

In this part, we will start explaining our design starting from Main Thread. We need to do some changes to the Main Thread to be suitable with our objectives, multithreaded web server. The Main Thread that we have is doing setup the socket, accepting connection, sending request using HTTP protocol to the web server and then delete the requests to go back and accept new connection request and so on. Also, in overload situation, the web server will stop responding when it receives a relatively small number of concurrent requests. But what we will do to be a multithreaded server are, it will keep doing setup the socket and accepting the connection as it, but we will store the requests in Circular Queue for the threads to be able to serve the requests. Also, the Main Thread will create A Monitor Thread, and this will be explained in point 4.2.

The new Main Thread design will handle overload situations which are, first, if there is enough space in buffers, then the web server will work normally. Second, if the buffers

are full, then the user has two options to handle this problem by specifying it in the command line. The two options are, first, the default one Block (BLCK), the Main Thread will block until the buffers becomes available. For the second one Drop\_head (DRPH), the main thread will drop the oldest request in the queue that is not currently being processed by a thread and add the new request to the end of the queue.

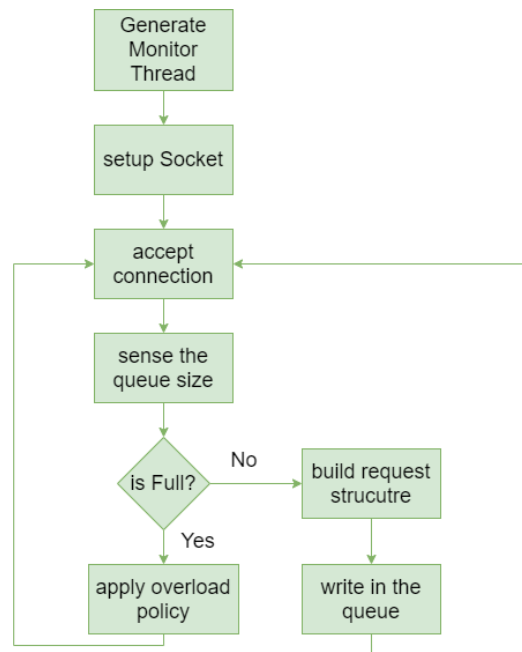


Figure 2: Main thread flowchart

### 1.4.2 Monitor thread

For the Monitor Thread, it will initialize an object from Thread Pool to gives it start sign to create Workers Threads. Also, Monitor Thread will monitor the live number of workers, report any abnormal conditions as they occur in the server, handle a termination signal (Ctrl-c) when it comes from the keyboard and clean-up before normal termination.

Table 1 UML for montor Thread

Monitor Thread
- s: ServeWebRequest
<<constructor>> ThreadPool() +worker_thread_number() +is_abnormal_condition() +termination_signal() +cleanup()

1. **worker\_thread\_number()** method, it will check the number of alive threads so, when it find one dead thread it will gives order to Thread Pool to replace it.

2. **is\_abnormal\_condtion()** method, it will check if there is abnormal condition, when it find it, it will report it to stdout.

3. **termination\_signal()** method, it will keeps seeking for Ctrl-c signal to terminate all threads in the thread-pool, terminate the monitor thread, destroy all semaphores, clean-up the memory and terminate gracefully.

4. **Cleanup()** method, to clean-up before normal termination.

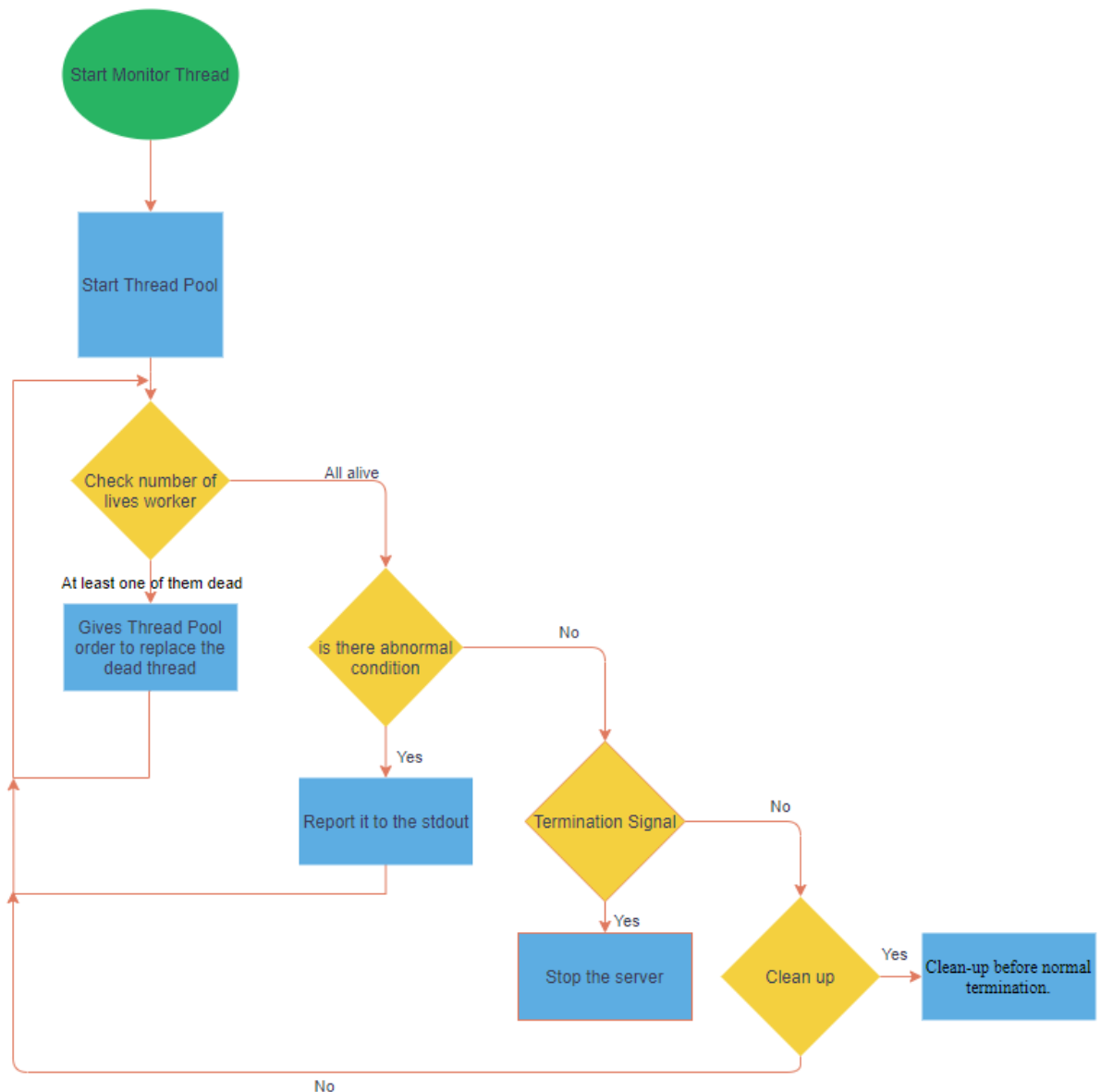


Figure 3 follow chart for monitor thread.

### 1.4.3 Thread Pool

This class will enhance our webserver performance greatly. By make N threads to serve requests. Thread pool will be initialized by the monitor thread using thread pool constructors. There are two constructors one with parameter that represent number of worker threads. And one parameter less using the default value of N threads. We called those thread as workers.

Table 2 thread Pool UML

Thread Pool
- workers: worker[] - s: ServeWebRequest - nworker:int
+ <<constructor>> pool() + <<constructor>> pool(nworker:int) + set(s: ServeWebRequest) + threads_states():State + mintain_threads() + cleanup()

#### Fields:

1. **Workers:** Array of worker thread that server requests taken from CircualrQueue.
2. **s:** same reference of ServeWebRequest in main passed from monitor thread and it will be passed to workers.
3. **nworker:** represent number of worker threads in thread pool to server requests.

#### Methods:

1. **Set():** used by monitor thread to pass the reference for ServeWebRequest which initialized in main Thread.
2. **Threads\_states():** return thread state to be used by monitor thread and maintenance.
3. **Mintain\_threads():** called by monitor threads to replace dead threads and maintain the nworker specified by server administrator.
4. **Cleanup():** used to interrupt worker threads and this make it in interrupted state.

### 1.4.4 Worker Thread

Worker is class that inherit Thread class. There will be N workers instilled by Thread pool constructor. Worker is responsible of serving one request at time by taking one request object from CircualrQueue Serve that request using ServeWebRequest object (one instance shared between all worker instances). finally removing request object taken from CircualrQueue.



Table 3 worker thread UML

Worker Thread
- s: ServeWebRequest - c: CircularQueue - r Request - file: web-server-log
+ <<constructor>> worker(s: ServeWebRequest, c:CircularQueue, file: web-server-log) - redirect_std(file: web-server-log) + get_request():request + send_respones() + run()

**Fields:**

1. **s:** one instance of ServeWebRequest shared between all workers instances used to serve request in send\_response() method.
2. **c:** CircularQueue reference used in to get request in get\_request() method.
3. **r:** Request object to hold the request taking from CircularQueue.

**Methods:**

1. **Worker (s: ServeWebRequest, c:CircularQueue, file: web-server-log):** Constructor will take ServeWebRequest and CircularQueue and web-server-log.
2. **Redirect\_std(file):** redirect stdout and stderr to log file it will be called in constructor worker.
3. **get\_request():** uses c(CircularQueue reference) to get one request from it.
4. **send\_respones():** uses s(ServeWebRequesut) to serve request.
5. **run():** executed when we start the thread in Thread pool constructor. It will get\_request() then send\_response() in infinity loop until interrupted by monitor thread using pool thread method cleanup()

**1.4.5 Circler Queue data structure**

The CircularQueue class is a generic class that provide First-in-First-out (FIFO) utilities. It is a thread safe data structure using the Semaphore utilities so that protection is handled internally. Choosing CircularQueue as the data structure that wraps all the request is because it follows the FIFO concept and the elements can be inserted and deleted with  $O(1)$ . Although linked list provides  $O(1)$  insertion and deletion, java arrays implementation is much efficient in terms of speed and memory usage. The instances for the class are as follow:

1. **currentSize:** private integer value represents the current number of elements in the queue.
2. **circularQueueElements:** private normal arrays handle and contains all the elements.

3. **maxSize**: private integer value represents the maximum allowed number of elements in the queue.
4. **front**: private integer value point to the front of the queue.
5. **rear**: private integer value points to the rear of the queue.
6. **mutex**: private Semaphore object for the queue protection.

These instances are manipulated using the following methods:

1. **CircularQueue<E>(size:int)**: public constructor to initialize the instances.
2. **enqueue(item:E)**: public method to add item at rear of the queue.
3. **dequeue()**: public method to return and remove item from the front of the queue.
4. **isFull()**: public method return True if the queue is full.
5. **isEmpty()**: public method returns True if the queue is empty.
6. **getSize()**: public method return the current number of elements in the queue..

Table 4 CircualrQueue UML

CircularQueue<E>
- currentSize:int - circularQueueElements:E[] - maxSize:int - front:int - rear:int - mutex: Semaphore
+ <<constructor>> CircularQueue<E>(size:int) + enqueue(item:E) + dequeue():E + isFull():bool + isEmpty():bool + getSize():int

#### 1.4.6 Request class

There are two data types that represent client request (Socket ClientSokcet, RequestNumber). But we want to make it as one instance to added to CircualrQueue. Request class will combine them in one data type called Request. Worker thread will take ClientSokcet and RequestNumber from Request instance using following utilities:

1. **get\_ClinetSocket()**: return ClientSokcet.
2. **Get\_RequestNumber()**: return int value represent request number.

Table 5 Request UML

Request
- RequestNumber:int - ClinetSocket: <b>Socket</b>
<<constructor>> Request(ClinetSocket: <b>Socket</b> , RequestNumber:int) + get_ClientSocket():Socket + get_RequestNumber():Int

### 1.5. Project Schedule

The project is consisting of three phases, Design, Implement and benchmark. As shown in the schedule table, the design phase is completed, and the implementation phase is under progress. Some dependences for some of the tasks have been defined.

Table 6 project timeline

Phase	Task	Status	Dependences
Design	Specify the server requirements.	Completed	N/A
	Specify main thread responsibilities.	Completed	N/A
	Specify monitor thread responsibilities.	Completed	N/A
	Specify the needed data structure and classes.	Completed	N/A
	Specify the utilities for all the classes.	Completed	N/A
Implementation	Implement and test the CircualrQueue	Completed	N/A
	Implement and test the Request structure	Completed	N/A
	Implement and test the main thread	Completed	CircualrQueue, Request structure
	Implement and test the Worker threads	Completed	CircualrQueue, main thread
	Implement and test the ThreadPool	Completed	Worker threads

	Implement and test the monitor thread	Completed	ThreadPool, main thread
	Test any starvation or deadlocks	Completed	All
Benchmarking	Benchmarking the server	Completed	All
	Improve any bottle necks in the design.	Completed	Benchmarking

## 1.6. Team Schedule

The project is consisting of three phases, Design, Implement and benchmark. As shown in the schedule table, the design phase is completed, and the implementation phase is under progress. Some dependences for some of the tasks have been defined.

Table 7 work distribution

Task	Assignee
Implement and test the CircularQueue	Mohammed
Implement and test the Request structure	Khalid
Implement and test the main thread	Khalid
Implement and test the Worker threads	Ahmad
Implement and test the ThreadPool	Ahmad
Implement and test the monitor thread	Mohammed
Test any starvation or deadlocks	All

## 1.7. Conclusion

In conclusion, the project is to improve a single thread server performance by modifying the design to multithread architecture. In order to achieve that objective, many design modules are have been illustrated such as CircularQueue, ThreaPool, Request structure, Worker Thread, Monitor thread and Main thread. All the parts must communicate properly, and the shared data must be mutual excluded. This step is done using the Semaphore which is implemented inside the CircularQueue class. All the worker threads will be trying to access CircularQueue and read request and at the same time the main thread trying to access it to write on it, so a synchronization algorithm using semaphores is done.

# Chapter 2 Benchmark Test

## 2.1. Introduction

The benchmark test will test the performance of the server in different cases. The aim is to benchmark the different overload policies and to find the optimal server settings. Thought that benchmark, three different tests has been conducted; test the BLCK policy and find the optimal set up, test the performance of the DRPT policy with the optimal set up and test the performance of the DRPH policy with the optimal set up. All the tests have been tested on the same device with the same enabled logical cores and same network interface.

## 2.2. Objectives

1. Find the maximum request rate with error free.
2. Find the optimal settings gives replay time less than 1000ms and response time less than 800ms [1].
3. Test the overload policies with the optimal error-free setting.

## 2.3. Device Specifications

1. 24 Logical Cores.
2. 10 giga byte ethernet interface.

## 2.4. Results

### 2.4.1. Optimal Error-free Settings

In this section, a test cases would be conducted in order to find the optimal error-free settings without violating objective 2. All the test was conducted in the device specifications illustrated earlier in section 3. All the test cases are using the BLCK policy. The cases from Case#1 through Case#12 shows different test results. Each time the server success with the initial request rate without errors, a step value to the request rate will be added and tested. Once an error occurs in any of the request rates, the previous error-free request rate will be tested again with more time to make sure that its error-free. The unique colored row in each test illustrates the maximum error-free request rate.

### Case#1:

In this test case, the request rate at 110 started to show errors which indicates that the server with the pool size of 10 and buffer size of 10 reached its limitation. The Avg reply time and response time is clearly increasing by increasing the request rate due to the limitation of the available server bandwidth for each user.

Test Total Time (minutes)	Pool Size	Buffer Size	Request Rate	Total Number of Requests	Number of Errors	Total Number of Replies	Avg Reply Time(ms)	Approximate Response (ms)
1	10	10	100	8,703	0	8,703	528	420
1	10	10	105	8,617	0	8,617	559	440
5	10	10	105	44,613	0	44,613	564	436
1	10	10	110	8,517	0	8,517	600	470
5	10	10	110	44,936	3	44,933	594	463
1	10	10	115	8,673	3	8,670	618	550
1	10	10	120	8,695	8	8,687	628	550

### Case#2:

In this test case, the request rate at 110 started to show errors which indicates that the server with the pool size of 10 and buffer size of 14 reached its limitation. The max-rate increased from the previous test to 110 due to the increasing in the buffer size.

Test Total Time (minutes)	Pool Size	Buffer Size	Request Rate	Total Number of Requests	Number of Errors	Total Number of Replies	Avg Reply Time(ms)	Approximate Response (ms)
1	10	14	105	8,499	0	8,499	561	475
1	10	14	110	8,661	0	8,661	589	480
5	10	14	110	44,589	0	44,589	596	479
1	10	14	115	8,627	1	8,626	633	520

### Case#3:

In this test case, the request rate at 115 started to show errors which indicates that the server with the pool size of 10 and buffer size of 18 reached its limitation. The max-rate at this test case is the same as the previous one since the increasing in the buffer size in only 4 additional slots.

Test Total Time (minutes)	Pool Size	Buffer Size	Request Rate	Total Number of Requests	Number of Errors	Total Number of Replies	Avg Reply Time (ms)	Approximate Response (ms)
1	10	18	110	8,472	0	8,472	594	473
5	10	18	110	44,565	0	44,565	580	476
1	10	18	115	8,419	2	8,417	636	520

#### Case#4:

In this test case, the request rate at 120 started to show errors which indicates that the server with the pool size of 10 and buffer size of 18 reached its limitation. The request rate at this test gives avg reply time and response time higher than the previous one since the number of the user higher because the buffer size allowed more users to be accepted.

Test Total Time (minutes)	Pool Size	Buffer Size	Request Rate	Total Number of Requests	Number of Errors	Total Number of Replies	Avg Reply Time (ms)	Approximate Response (ms)
1	10	22	110	8,632	0	8,632	579	480
1	10	22	115	8,419	0	8,419	630	538
5	10	22	115	43,523	0	43,523	644	532
1	10	22	120	8,452	2	8,450	662	550

#### Case#5:

In this test case, 120 was the maximum error-free request rate. The buffer size available slots are directly related to the maximum error-free request rate.

Test Total Time (minutes)	Pool Size	Buffer Size	Request Rate	Total Number of Requests	Number of Errors	Total Number of Replies	Avg Reply Time (ms)	Approximate Response (ms)
1	10	26	120	8,769	0	8,769	638	530
5	10	26	120	44,210	0	44,210	667	532
1	10	26	125	8,593	3	8,590	681	573

#### Case#6:

In this test case, 125 was the maximum error-free request rate. At 130 request rates with a test of 1 minute long, no errors were occurred. With additional testing time request rate of 130 failed to have error-free.

Test Total Time (minutes)	Pool Size	Buffer Size	Request Rate	Total Number of Requests	Number of Errors	Total Number of Replies	Avg Reply Time (ms)	Approximate Response (ms)
5	10	30	125	43,442	0	43,442	711	560
1	10	30	130	8,418	0	8,418	737	610
5	10	30	130	44,412	2	44,410	699	612

#### Case#7:

In this test case, 160 was the maximum error-free request rate. The avg reply time and response time is still shows a relation between the buffer size and request rate.

Test Total Time (minutes)	Pool Size	Buffer Size	Request Rate	Total Number of Requests	Number of Errors	Total Number of Replies	Avg Reply Time (ms)	Approximate Response (ms)
1	10	60	130	8,297	0	8,297	720	650
1	10	60	160	8,298	0	8,298	946	770
5	10	60	160	42,932	0	42,932	961	890
1	10	60	165	8,177	0	8,177	952	850
5	10	60	165	42,598	35	42,563	995	840
1	10	60	170	8,182	4	8,178	990	850

#### Case#8:

In this case, Pool size increase from 10 to 12 and it shows better improvement for request rate than Case#2 which buffer size increased from 10 to 14. This indicate of better usage of available 24 logical cores in our server.

Test Total Time (minutes)	Pool Size	Buffer Size	Request Rate	Total Number of Requests	Number of Errors	Total Number of Replies	Avg Reply Time (ms)	Approximate Response (ms)
1	12	10	60	8,557	0	8,557	271	180
1	12	10	80	8,362	0	8,362	395	300
1	12	10	100	8,406	0	8,406	521	420
1	12	10	110	8,285	0	8,285	598	500
5	12	10	115	43,333	0	43,333	632	520
1	12	10	120	8,523	1	8,522	647	550

#### Case#9:

In this case, Pool size increase from 12 to 14. Request rate increase from 115 to 120. Which is clear improvement from previous Case#8. On the other Hand, Avg Replay time increase from 632 ms to 670 ms and Response time from 520 ms to 530 Ms. This number is still satisfying our second objectivate.

Test Total Time (minutes)	Pool Size	Buffer Size	Request Rate	Total Number of Requests	Number of Errors	Total Number of Replies	Avg Reply Time (ms)	Approximate Response (ms)
1	14	10	120	8,200	0	8,200	653	530
5	14	10	120	42,022	0	42,022	670	530
1	14	10	125	8,335	5	8,330	683	550

#### Case#10:

In This case, we are experimenting by increasing buffer size to almost three times the previous case 10 to 28. This test does not show that much of different from the previous Case#9. So, we decide to increase Pool size in next Cases.



Test Total Time (minutes)	Pool Size	Buffer Size	Request Rate	Total Number of Requests	Number of Errors	Total Number of Replies	Avg Reply Time (ms)	Approximate Response (ms)
1	14	28	120	8,419	0	8,419	631	500
5	14	28	135	42,738	0	42,738	753	620
1	14	28	140	8,427	0	8,427	756	650
5	14	28	140	42,601	6	42,595	797	650

#### Case#11:

Second Object sit a clear upper bound for Avg Replay time and Response time in this test. we hit that ceiling so we cannot increase the Pool size and buffer size any more than these numbers. Furthermore, we are aiming to find the optimal pool size to keep the request rate and minimize the avg replay and response time.

Test Total Time (minutes)	Pool Size	Buffer Size	Request Rate	Total Number of Requests	Number of Errors	Total Number of Replies	Avg Reply Time (ms)	Approximate Response (ms)
1	20	32	170	8,491	0	8,491	902	730
5	20	32	170	42,549	0	42,549	953	740
1	20	32	175	8,356	14	8,342	941	800

#### Case#12:

This case gives us the best result yet with highest request rate 175 and one of the lowest response time 700 ms and Avg replay time 971 ms.

Test Total Time (minutes)	Pool Size	Buffer Size	Request Rate	Total Number of Requests	Number of Errors	Total Number of Replies	Avg Reply Time (ms)	approximate Response (ms)
5	28	20	175	40,860	0	40,860	971	700
5	28	20	180	40,773	11	40,762	966	750

#### Case#13:

In this case, we have better Avg time but then Case#12 but lower request rate without errors. So, we eliminate it and take Case#12 as our optimal case.

Test Total Time (minutes)	Pool Size	Buffer Size	Request Rate	Total Number of Requests	Number of Errors	Total Number of Replies	Avg Reply Time (ms)	Approximate Response (ms)
5	30	18	170	40,534	0	40,534	944	700
5	30	18	175	40,922	4	40,918	973	700

### 2.4.2. Optimal Error-free Settings Analysis

As shown in figure 4, there are a linear relation between the buffer size and the request rate which give an indication of where the optimal setting will be. If the optimal pool size is founded, the best buffer size would be the maximum size that has error-free request rate.

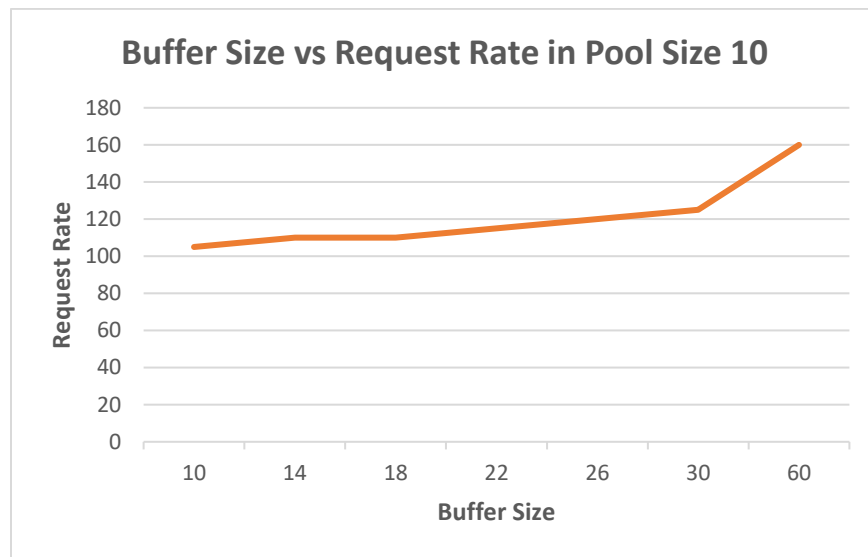


Figure 4: Buffer size and request rate relation

Figure 5 shows the relation between the results of the pool size and buffer size with the avg replay time. Minimum avg replay time is with the minimum buffer size as explained earlier. If the only factor for choosing the optimal setting is the avg reply time, then a buffer size and poll size of 10 would be the choice. But this no the case, where the objective is to find a minimal replay time and response time with the maximum request rate.

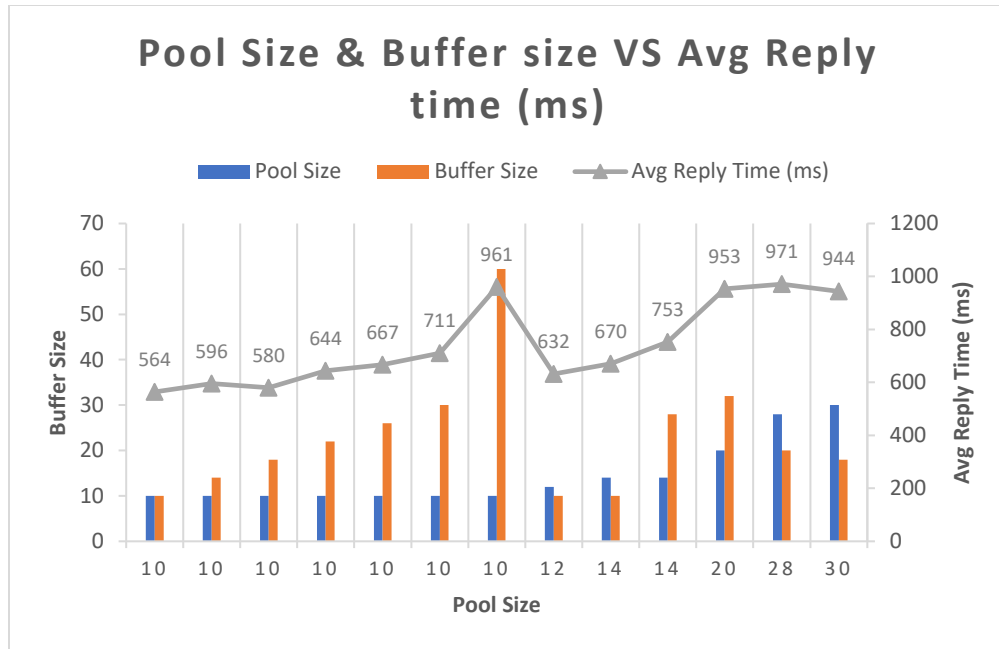


Figure 5: Pool size and Buffer size Vs Avg Reply time

Figure 6 shows same as figure 5 but with the response time. The relation between the response time and the pool and buffer size is the same as the relation with the reply time. The same case is valid here whether if the only factor is the response time. But this is not the case, and the max-rate must be considered.

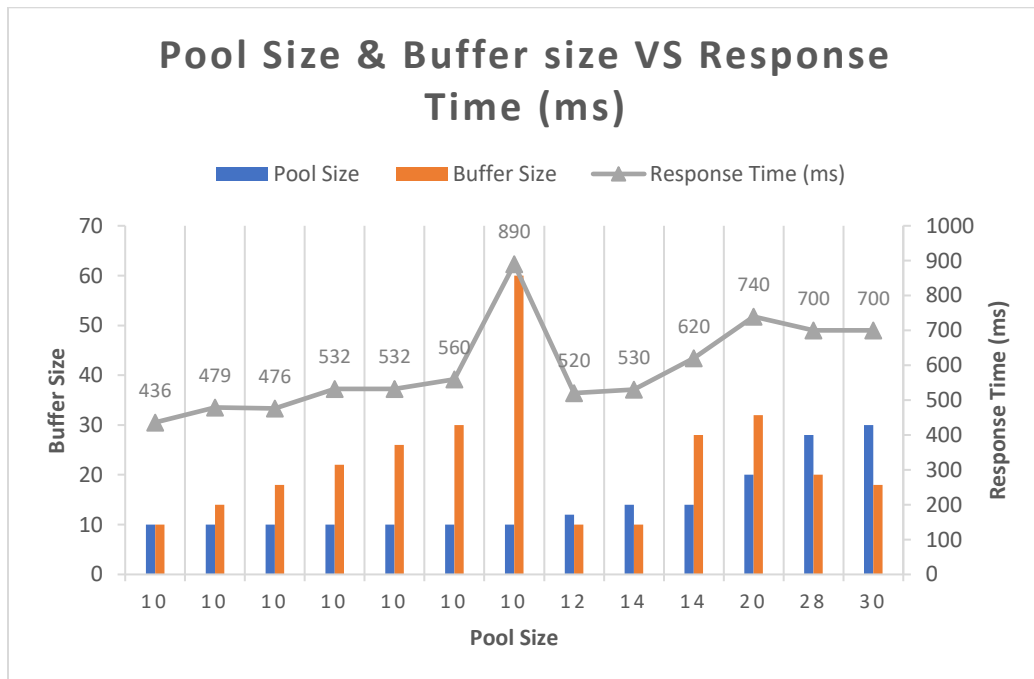


Figure 6: Pool and Buffer size Vs Response time

Figure 7 shows the relation between the pool and buffer size pairs. The best maximum rate occurred with pool size 28 and buffer size 20 giving a 175 request rate. When considering this pool and buffer size in figure 5 and 6, it has achieved the specified objectives to have replay time less than 1000ms and response time less than 800ms.

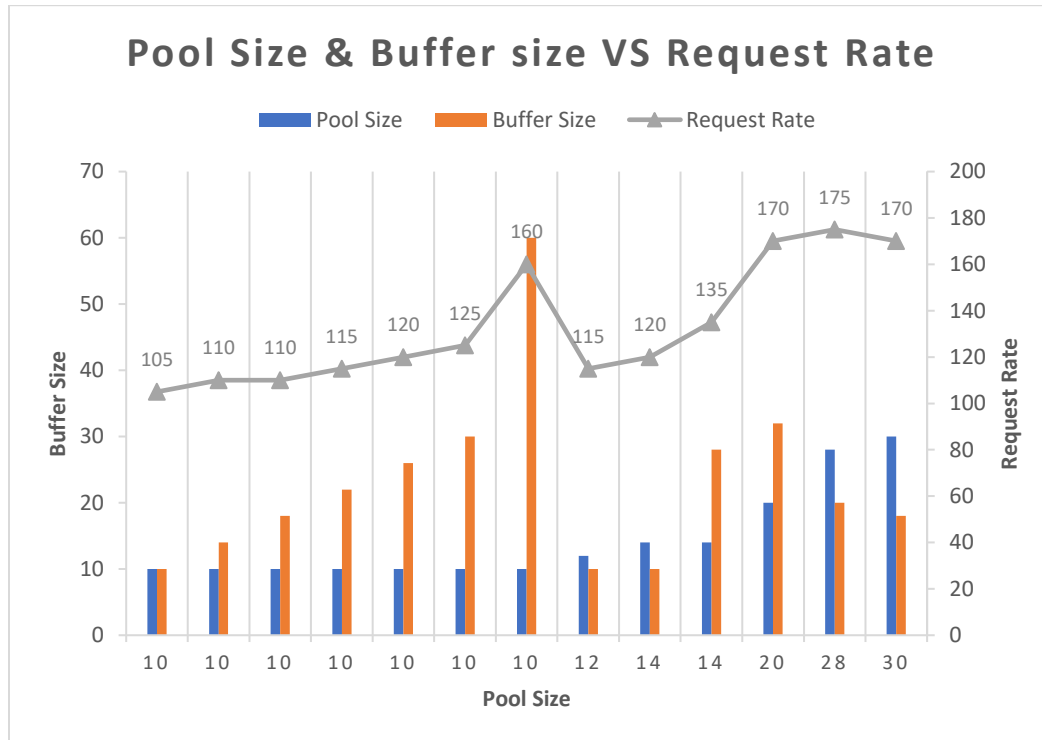


Figure 7: Pool and Buffer size Vs Request rate

## 2.4.3. Overload Policies Performance

### 2.4.3.1 DPRT

DRPT Drop Tail Policy it achieves error free 35 request rate and Average Reply time 127 ms and Response time 75 ms. It has a flat reply time from 4 active users until 12 active users then it keeps increasing. Compare to basic webserver it has better reply time and response time. Last figure shows the same test after increasing request rate to 44 and the error rate increase to 0.04%.

Test Total Time (minutes)	Pool Size	Buffer Size	Request Rate	Total Number of Requests	Number of Errors	Total Number of Replies	Avg Reply Time (ms)	Approximate Response (ms)
1	28	20	35	8,192	0	8,192	137	75
5	28	20	35	42,874	0	42,874	127	75
1	28	20	40	8,377	1	8,376	159	80

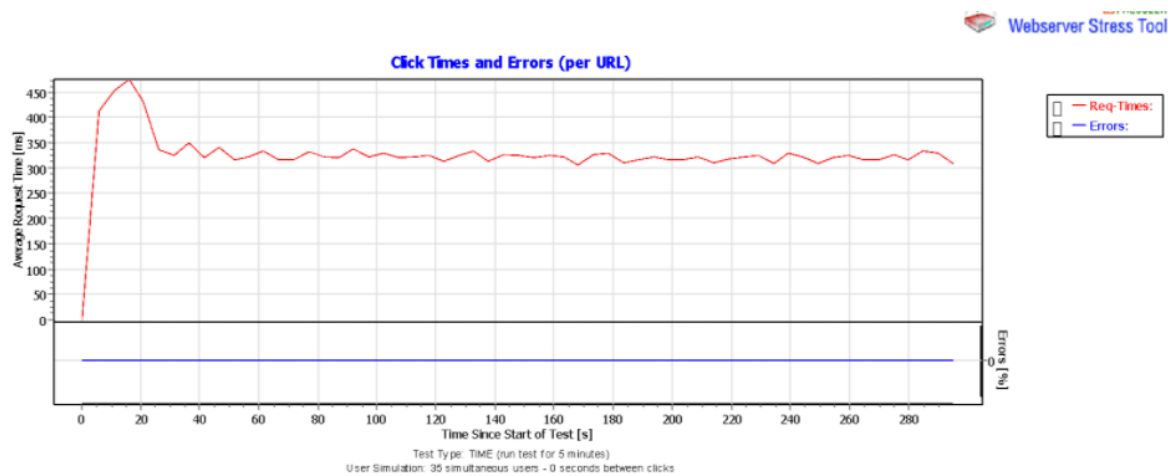


Figure 8: basic server replies time.

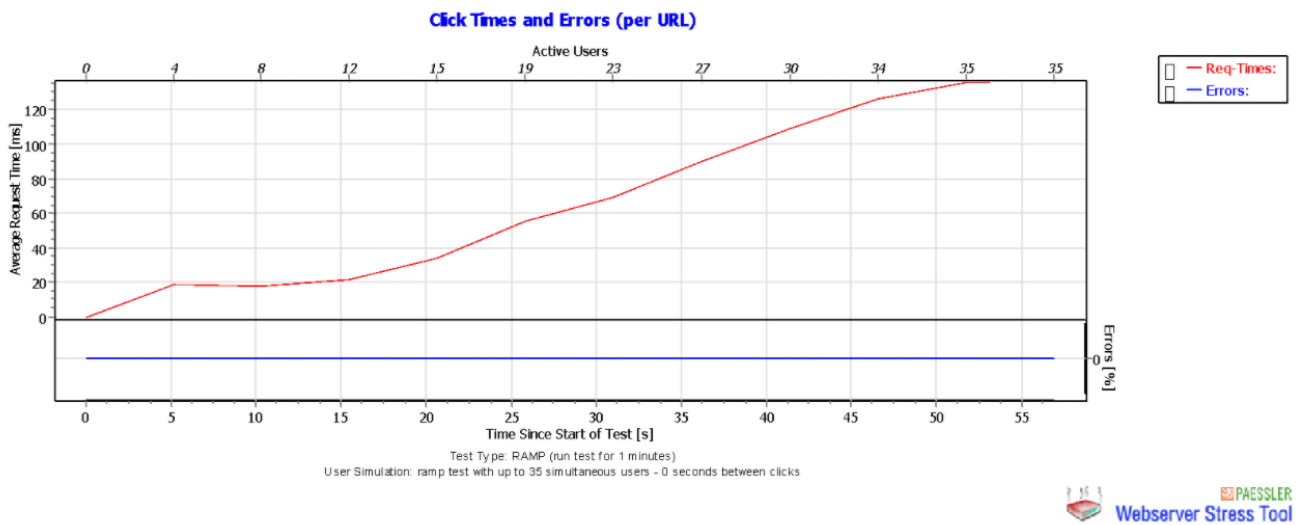


Figure 9: request rate at which reply-time latency increases.

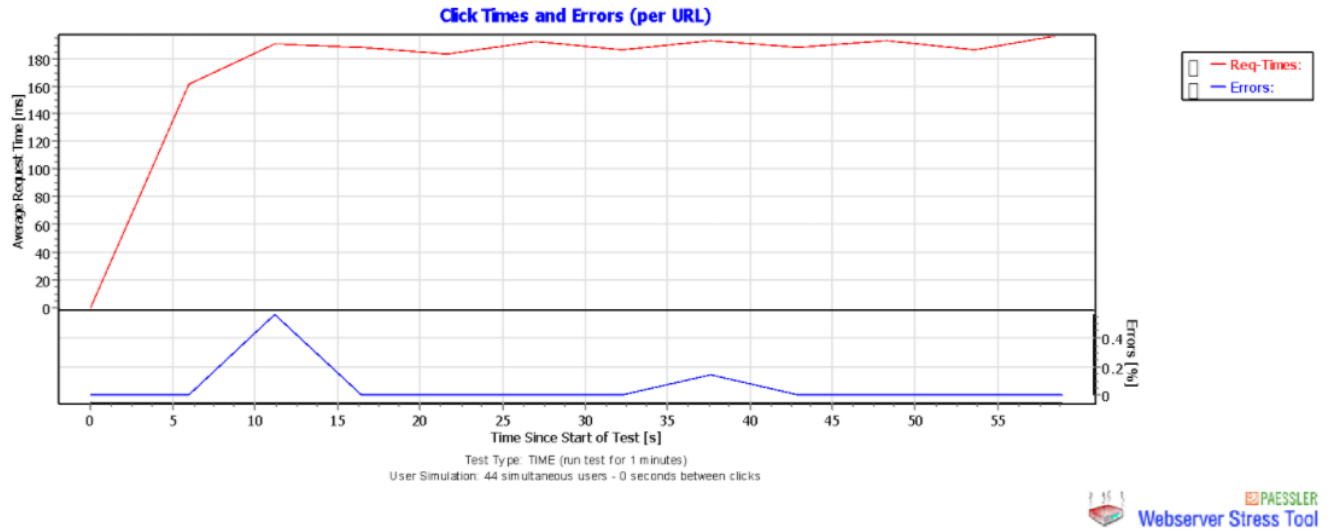


Figure 10: error rate and reply time server operates 25% above the maximum error-free rate.

#### 2.4.3.2 DPRH

The DPRH policy achieved error-free request rate at 30. As a comparison between the multithreaded web server with the DPRH policy and the basic server, the basic server achieved high avg replay time and response time. On the other hand, the multithreaded web server achieved lower reply time and response time. As figure 12 showing, the reply latency shows a flat relation between 4 to 10 active users. After 10 it starts to increase linearly. Figure 13 shows the server reply latency and errors rate when the server operates with 25% above the maximum error-free rate.

Test Total Time (minutes)	Pool Size	Buffer Size	Request Rate	Total Number of Requests	Number of Errors	Total Number of Replies	Avg Reply Time (ms)	Approximate Response (ms)
1	28	20	30	8,267	0	8,267	112	60
5	28	20	30	44,009	0	44,009	108	60
1	28	20	35	8,122	1	8,121	140	75

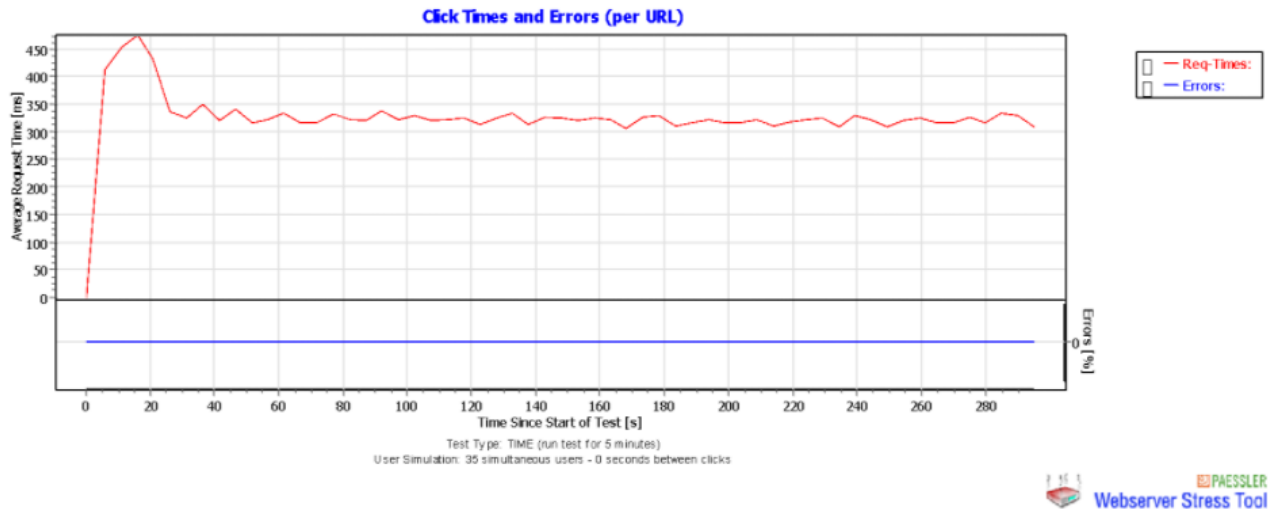


Figure 11: basic server replies time.

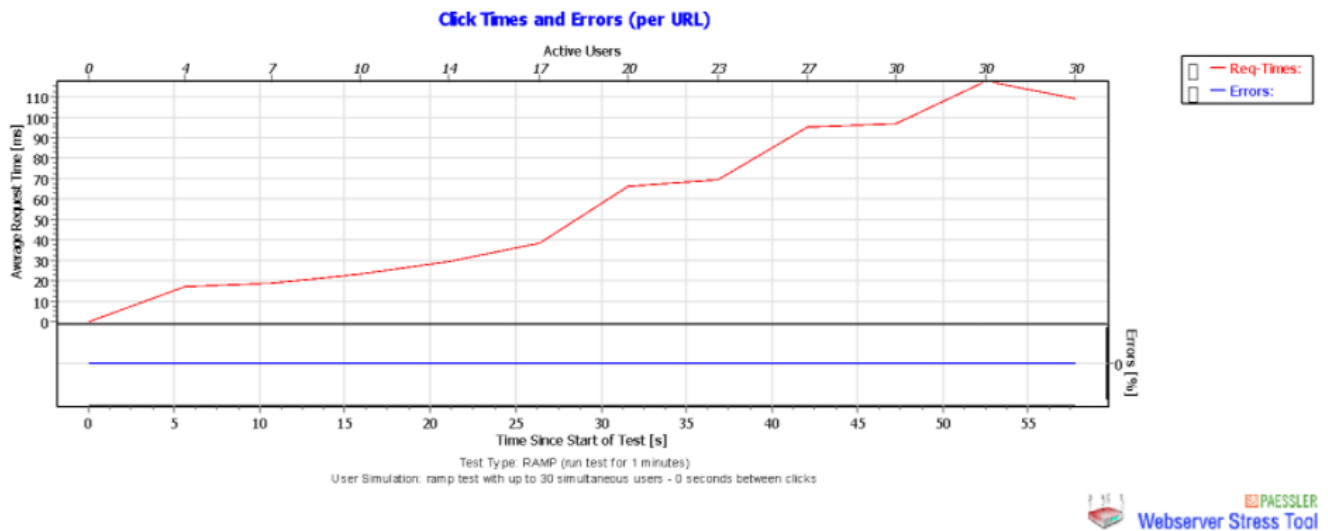


Figure 12: request rate at which reply-time latency increases.

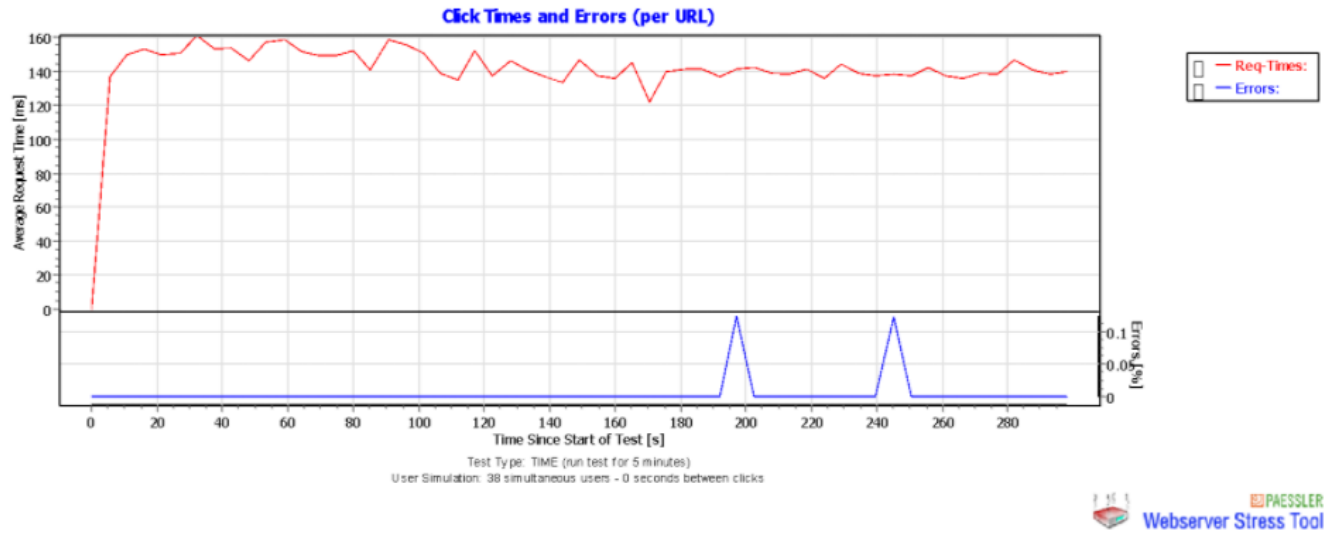


Figure 13: error rate and reply time server operates 25% above the maximum error-free rate.

## 2.5. Suggestions

- 1- Overload Policy that track request and serve fastest request first.
- 2- Increase the network bandwidth.
- 3- Increase the number logical cores.
- 4- Improve the secondary memory latency.



## References

1- Why server response time is important. (n.d.). Retrieved April 20, 2021, from <https://www.hrank.com/why-server-response-time-is-important#:~:text=What%20is%20average%20server%20response,more%20than%20800ms%20is%20slow>