Modeling Website Performance Under Different Traffic Conditions

Group Member: Christopher Xu, Daming Wang, Wei Wang

Table of Contents

1.0 Problem Description....4

2.0 Simulation Model....4

3.0 Simulation Goals and Parameters.... 5

3.1 Goals.... 5

3.2 Parameters.... 6

4.0 Methodology.... 7

4.1 How is Java used to process the simulation?....7

4.2 Simulation Main Flow....11

4.3 Simulation Setup....11

4.4 Collect the stats.... 12

5.0 Analysis....13

5.1 100 Lines of Data Analysis.... 14

5.2 10,000 Lines of Data Analysis.... 17

5.3 Difference of Priority Method Analysis....22

6.0 Discussion and Conclusion....22

6.1 Conclusions.... 22

Conclusion 1: The relationship between the number of concurrent users and the total waiting time is nearly linear but not strictly linear....23

Conclusion 2: The relationship between concurrent users and total service time is linear.. 24

Conclusion 3: The relationship between the number of concurrent users and average waiting time shows an exponential distribution....24

Conclusion 4: The relationship between the number of concurrent users and the average service time obeys the following distribution:....25

Conclusion 5: The relationship between total waiting time and total service time is linear, but with some deviation on the right side of the linear line with larger x values....26

Conclusion 6: The average waiting time and service time ratio are 0.33 and 3, which fits the earlier parameters....26

Conclusion 7: The relationship between users and maximum workload shows a stepped relationship with 480, 750, and 3500 users at each step....26

Conclusion 8: The difference between the two baselines, different and equal priority methods, is insignificant....27

6.2 Suggestions....27

Table of Figures

Figure 1 - Simulation Model Design....4

Figure 2 - Figure 2: State Transition Diagram of a Thread....7

Figure 3: Shell Script Code of Simulation Set Up....12

Figure 4. Codes that output the output variables to a CSV file...13
Figure 5. Concurrent Users and Total Waiting Time for 100 data…………………………………14
Figure 6. Concurrent Users and Total Service Time for 100 data…………………………….……15
Figure 7. Concurrent Users and Average Waiting Time for 100 data……………………………..15
Figure 8. Concurrent Users and Average Service Time for 100 data……………………………..16
Figure 9. Total Waiting Time and Total Service Time for 100 data…………………………………16
Figure 10. Average Waiting Time and Average Service Time for 100 data………………………17
Figure 11. Concurrent Users and Total Service Time for 10k data ……………………….……18
Figure 12. Concurrent Users and Total Waiting Time for 10k Data ……………………………..…19
Figure 13. Concurrent Users and the Maximum Workload for 10k data…………………………..19
Figure 14. Concurrent Users and Average Waiting Time for 10k data……………………………20
Figure 15. Concurrent Users and the Average Service Time for 10k data ..……………………..20
Figure 16. Total Waiting Time and the Total Service Time for 10k data …………………………..21
Figure 17. Average Waiting Time and the Average Service Time for 10k data …..……………..21
Figure 18. Relationship Among Concurrent Users, Total Waiting Time Difference, and Maximum Workload Difference……………………………………………………………………………………22

Table of Tables

Table 1: Simulation output variables ………………………………………………………………...6

1.0 Problem Description

The project aims to simulate an Internet server group that handles concurrent users by queuing First-In-First-Out client requests. The desired outcome is demonstrating the numerical relation between servers and concurrent users. This project is coded in Java. The purpose is to collect enough statistics to show the relation between the parameters of servers and the number of concurrent users.

2.0 Simulation Model

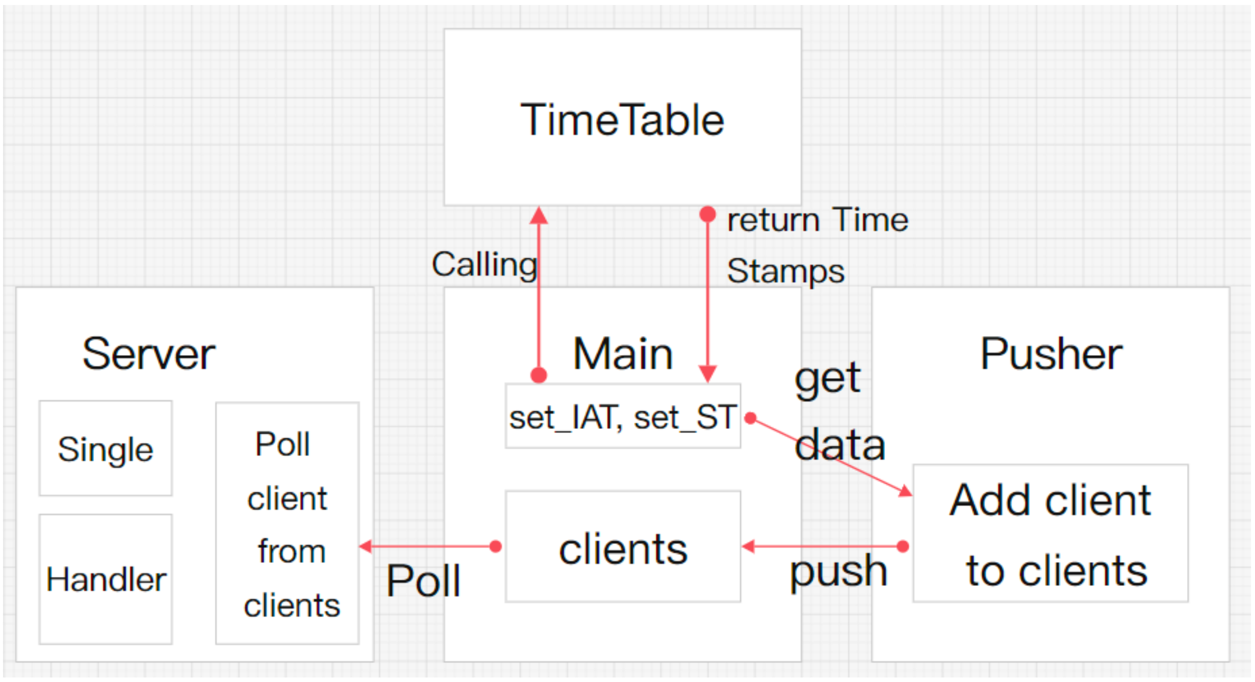


Figure 1: Model Prototype

The model has three primary nodes: A request pusher as requests sent from clients, thread pools as servers, and the main algorithm. In addition, we have two support nodes: A client object and A timetable.
There are two baselines: One is to allocate the requests by priority of each server, and a server with lower priority will have a lower chance of handling the request. The other one is to process the requests with the same priority. Both strategies have 12 handlers.

The ***main*** is the data transportation center. It generates Poisson process time of service time and inter-arrival time by calling methods in the TimeTable class and stores the data into two arrays: inter_arrival_array and service_time_array. The exponential distribution is based on a mean of pre-set lambda integer in the Main class. Because of the randomness introduced by the distribution, a client queue called “clients” must be in place to collect all the clients. (The programmer manually defines the parameter for exponential distribution mean). This function would create other threads, which will be introduced below to implement the simulation.

The ***pusher*** is a runnable object by implementing a Runnable interface, which means it can be run as a thread to add the client object into the client queue. The initial information of each client is from inter_arrival_array and service_time_array in the ***Main***. When the number of created users has reached the size that the user manually defined in the ***Main***, the pusher thread will shut down.

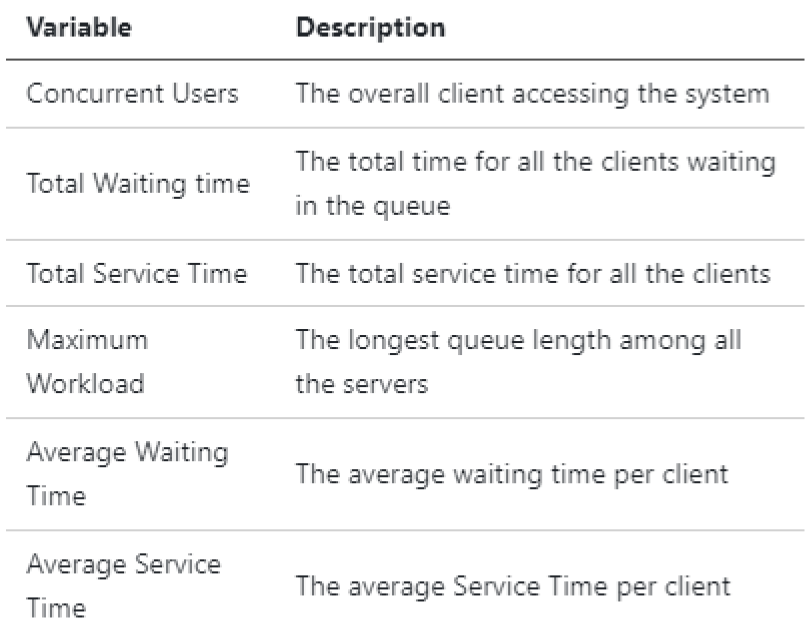
The ***Server*** has two “run()” algorithms, one algorithm is to create three thread pools (class Single in Server.java), and these three thread pools are ordered by their priorities(1, 5, and 10). Each thread pool is fixed and generates four threads(class Handler in Server.java) to handle client requests. The other algorithm is also to create three default priority thread pools with four threads for each pool(class Handler in Server.java), and all these 12 threads have the same opportunity to handle a client request if they are idle.

3.0 Simulation Goals and Parameters

3.1 Goals

The primary goal is to check whether the website performance is linearly related to the number of concurrent users. Ideally, in our null hypothesis, the server load should be approximately linearly related to the number of concurrent users. We used many different parameter sets in this simulation model to illustrate the dynamic relationships. The secondary goal is to compare the two performances between two baselines: different and equal priority servers. Additionally, the overarching goal was to find potential hidden relationships between the parameter sets and analyze any exciting discoveries. The output variables collected for analysis are described in Table 1.

Table 1: Simulation output variables.



Variable	Description
Concurrent Users	The overall client accessing the system
Total Waiting time	The total time for all the clients waiting in the queue
Total Service Time	The total service time for all the clients
Maximum Workload	The longest queue length among all the servers
Average Waiting Time	The average waiting time per client
Average Service Time	The average Service Time per client

3.2 Parameters

The simulation parameters are given below:

1. Size: The size of the simulation indicates the number of clients or requests that will be processed during the simulation. In the code attached, the concurrent users' size ranges from 1 to 5000. This simulation's total number of clients or requests should affect the overall server load and helps analyze how the system performs under different load.
2. Lambda_iat (Inter-Arrival Rate): This parameter determines the average number of clients arriving at the server. The inter-arrival times are modeled as an exponential distribution with a mean of 1/lambda_iat. In the provided code, lambda_iat is set to 2, meaning the average time between client arrivals is 0.5-time units. The inter-arrival time is essential to comprehending the dynamics of the queuing system and its ability to handle incoming requests.
3. Lambda_st (service time rate): Lambda_st represents the service time rate. It is the average rate at which the server processes requests. Similar to the arrival time rate, service times are modeled as an exponential distribution with a mean of 1/lambda_st. In our simulation, the service time rate is set to 3, which means the average service time is about 0.33-time units. The service time rate is critical to evaluate the server's processing capacity.
4. Clients: This parameter is the queue of the clients waiting in line. We used a BlockingQueue to implement it so that the server could process requests in the order they arrived.
5. Start_Time: This value is the time when the simulation started. It is used to calculate the arrival times.

4.0 Methodology

4.1 How is Java used to process the simulation?

The simulation was comprised of six Java source files. Four defined the simulation nodes, one defined the minimum process unit and the information that could be involved, and the last one described the two essential time stamps: inter-arrival time and service time of each process unit.

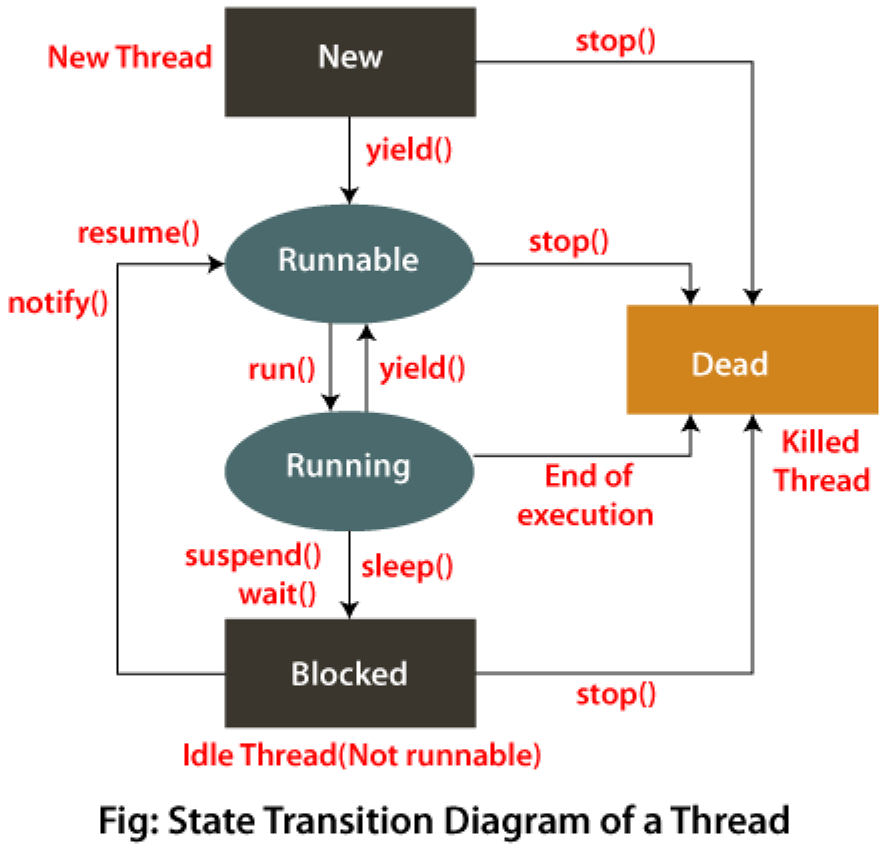


Figure 2: State Transition Diagram of a Thread

The files are listed below(all are included, in full, in Appendix 待排序)

- Pusher.java
- Main.java
- Server.java
- Server2.java
- TimeTable.java
- Client.java

The Pusher class implements the Runnable interface. The pusher is coded to get the following parameter from Main.java:

- Reference of Main.clients BlockingQueue
- Number of Concurrent Users

And taking data from:
- Main.inter_arriving_time_array
- Main.service_time_array
- Mian.arriving_time_array

The basic algorithm is to obtain this information to create a Client object which is defined in Client.java, then add the Client object into a BlockingQueue called "clients" in Main.java. After creating a new object, the Pusher thread will sleep, which means setting this thread into blocking status for the inter-arrival time of the Client object.

The Main class has eighteen fields, ten fields for support utility services; they are:
- int size
- final float lambda_iat
- final float lambda_st
- long[] inter_arriving_time_array
- long[] service_time_array
- long[] arriving_time_array
- BlockingQueue<Client> clients
- long start_time
- boolean step1
- boolean step_2

Furthermore, eight fields for other methods to call for final result calculation; they are:
- final AtomicLong concurrent_users_1
- final AtomicLong total_waiting_time_1
- final AtomicLong total_service_time_1
- final AtomicLong max_workload_1
- final AtomicLong concurrent_users_2
- final AtomicLong total_waiting_time_2
- final AtomicLong total_service_time_2
- final AtomicLong max_workload_2

The main algorithm is to create inter_arriving_time_array, service_time_array, and arriving_time_array for other threads to call. Then it instantiates a BlockingQueue as ArrayBlockingQueue with a length of a random number from 1 to 5000, which is also the allocated number of concurrent users. It is a BlockingQueue because our handlers will process multiple clients asynchronously. Furthermore, it needs to be asynchronous because a synchronized process will waste much time waiting for the calling back of results, and an asynchronous operation is more closed to reality.

Then start a Pusher and a Server to process the baseline1(priority) and generate the result of baseline1.

After finishing the baseline1, start a Pusher again and a Server2 to process the baseline2(non-priority) and generate the result of baseline2 and the difference between the two baselines.

The Server class implements a Runnable interface. The server is coded to handle the client in Main.clients. It has two subclasses:

- class Single extends Thread
- class Handler extends Thread

Moreover, the server has two parameters for its constructor:

- Reference of Main.clients as final BlockingQueue<Client> queue
- final int size

The basic algorithm is to create three Single threads with priorities 1, 5, and 10.
The single object also has two parameters, as same as the Server object.
Each Single thread creates a thread pool with four fixed Handler threads using ThreadPoolExecutor to process the client requests. It is ThreadPoolExecutor but not ExecutorServices because using ExecutorServices will soar the CPU utilization to around 100% and memory usage to about double. Then it will allocate the Client objects in Main.clients to its Handler threads if only some requests have been pushed and the Main.clients get empty. After all the client requests have been processed, Single will shut down its thread pool and generate the result of baseline1.

Handler object also has two parameters, as same as the Single object.
When handling a client request, it will poll a client instance from Main.clients and sleep the service_time of that client Object to simulate the processing time of handling a client request if there is nothing in the Main.clients, it will yield() back to the Runnable state from Running State.

The Server2 class implements a Runnable interface. The server is coded to handle the client in Main.clients. It has two subclasses:

- class Single2 extends Thread
- class Handler2 extends Thread

And the server has two parameters for its constructor:

- Reference of Main.clients as final BlockingQueue<Client> queue
- final int size

And one synchronized field:

- public static AtomicInteger count

The basic algorithm is to create three Single threads with default.
The single object also has two parameters, as same as the Server object.
Each Single thread creates a thread pool with four fixed Handler threads using ThreadPoolExecutor to process the client requests. Then it will allocate the Client objects in Main.clients to its Handler2 threads if only some users have been pushed and the Main.clients get empty. After all the client requests have been processed, it will shut down its thread pool and add one to Server2.count by count.incrementAndGet(). When the number of shut thread pools reaches 3, Single2 will generate the result of baseline2 and the difference between baseline1 and baseline2.

Handler2 object also has two parameters, as same as the Single2 object.
When handling a client request, it will poll a client instance from Main.clients and sleep the service_time of that client Object to simulate the processing time of handling a client request if there is nothing in the Main.clients, it will yield() back to the Runnable state from Running State.

The Client class is the object of describing a client request; it has six private fields:

- boolean serviced
- long service_time
- long service_start_time
- long inter_arrival_time
- long arrival_time
- long waiting_time
- double time_in_system

Its constructor method requires three parameters:

- long service_time
- long inter_arrival_time
- long arrival_time

And offers all the getters and setters to its fields.

The TimeTable class is to offering the Poisson Process numbers; it has three private fields:

- long[] inter_arriving_time_array
- long[] service_time_array
- long seed

Its constructor method requires three parameters:

- float lambda_iat
- float lambda_st
- long size

Then it calls two methods to finish the initialization:

- setInterArrivingTime(lambda_iat, size)
- setService_time_array(lambda_st, size)

By using these two methods:

```
private static int possionNumberGen(double lambda, double y){
        int x = 0;
         double cdf = PDF(x, lambda);
        while (cdf < y){
                x++;
                cdf += PDF(x, lambda);
        }
        return x;
}
private  static double PDF(double k, double lambda){
        double c = Math.exp(-lambda), sum = 1;
        for(int i = 1; i <= k; i++){
                sum *= lambda/i;
        }
        return sum * c;
}
```
To generate integer Poisson Process numbers.

4.2 Simulation Main Flow

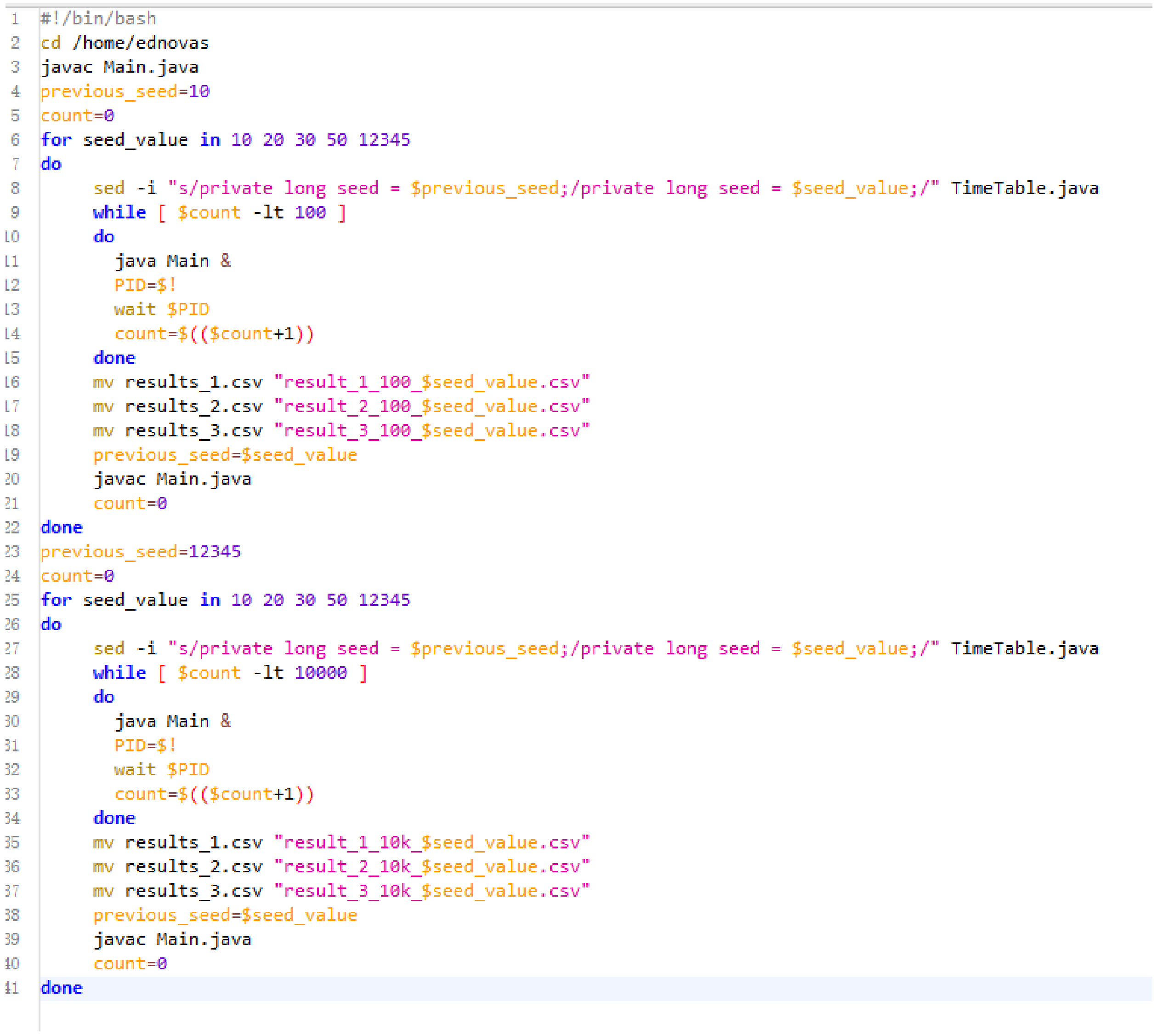
Step 1: We generate a random number between 1 and 5000 as the number of concurrent website users.
Step 2: Initialize the main algorithm, generating a random number from 1 to 5000 as the size of concurrent users on the website.
Step 3: Initialize the current time; it is the time when the server starts.
Step 4: Initialize the timetable; we assume that the number of visits per minute, represented in milliseconds, and the visit duration follow the Poisson arrival process.
Step 5: Prepare for the client queue: with the simulated size of concurrent users generated from the main.
Step 6: Initialize the servers and the pusher.
Step 7: The pusher sends the client requests into the client queue, then the server pulls these requests for handling.
Step 8: For implementing baseline1, the server will simulate three thread pools: high priority, medium priority, and low priority. Each thread pool has four Handler threads to handle the requests polled from Main.clients. After all the requests are processed, it will write these data as a CSV file to the output.
Step 9: For implementing baseline 2, the server will simulate three threads with the same priority. Each thread pool has four Handler threads to handle the requests polled from Main.clients. After all the requests are processed, it will write these data and the difference between baseline 1 and 2 as two CSV files to the output.
Step 10: Use Microsoft Excel to draw scatters for further comparison.

4.3 Simulation Setup

The program ran the following sets of simulations:

- The initial test was conducted on small runs of < 1000 concurrent users, and using breakpoints to debug the statements in the code may have issues. After the small number ran correctly, we added more concurrent users, up to 100,000, for extreme tests.
- Each of the 100 runs was done using a seed of 10, 20, 30, 50, and 12345 concurrent users per run.
- Each of the 10,000 runs was done using a seed of 10 concurrent users per run.

Using a shell script, we set up each simulation, shown in Figure 3.



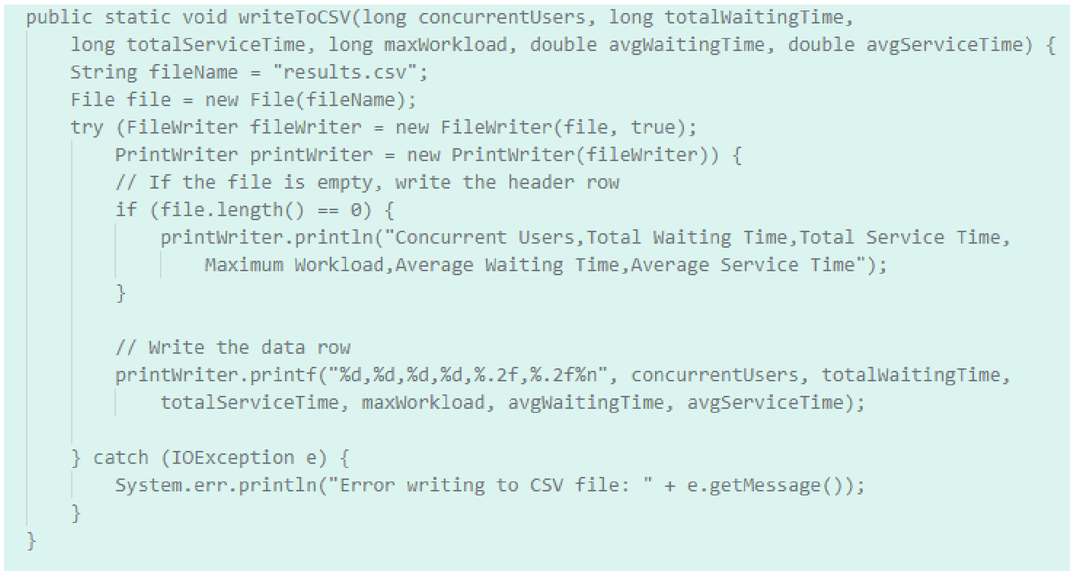
```bash
#!/bin/bash
cd /home/ednovas
javac Main.java
previous_seed=10
count=0
for seed_value in 10 20 30 50 12345
do
     sed -i "s/private long seed = $previous_seed;/private long seed = $seed_value;/" TimeTable.java
     while [ $count -lt 100 ]
     do
       java Main &
       PID=$!
       wait $PID
       count=$(($count+1))
     done
     mv results_1.csv "result_1_100_$seed_value.csv"
     mv results_2.csv "result_2_100_$seed_value.csv"
     mv results_3.csv "result_3_100_$seed_value.csv"
     previous_seed=$seed_value
     javac Main.java
     count=0
done
previous_seed=12345
count=0
for seed_value in 10 20 30 50 12345
do
     sed -i "s/private long seed = $previous_seed;/private long seed = $seed_value;/" TimeTable.java
     while [ $count -lt 10000 ]
     do
       java Main &
       PID=$!
       wait $PID
       count=$(($count+1))
     done
     mv results_1.csv "result_1_10k_$seed_value.csv"
     mv results_2.csv "result_2_10k_$seed_value.csv"
     mv results_3.csv "result_3_10k_$seed_value.csv"
     previous_seed=$seed_value
     javac Main.java
     count=0
done
```

Figure 3: Shell Script Code of Simulation Set Up

4.4 Collect the stats

The statistics collection was done in two parts. The first part is to write all the required output variables to the CSV file “results.csv”. In the run() function in the Server.java file, call this writeToCSV function, and once the function is called with the parameters delivered, the function will do the following:

1. Check whether there exists the “results.csv” file. If it exists, open the file and go next step
    a. If such does not exist such file, create one.
2. If there is no content in the file, write the first line for the table header
    a. If there is content, go to the next step directly
3. Write all the output variables generated by this simulation run.
4. If any IO exception is caught, throw an error.



```java
public static void writeToCSV(long concurrentUsers, long totalWaitingTime,
    long totalServiceTime, long maxWorkload, double avgWaitingTime, double avgServiceTime) {
    String fileName = "results.csv";
    File file = new File(fileName);
    try (FileWriter fileWriter = new FileWriter(file, true);
        PrintWriter printWriter = new PrintWriter(fileWriter)) {
        // If the file is empty, write the header row
        if (file.length() == 0) {
            printWriter.println("Concurrent Users,Total Waiting Time,Total Service Time,
                Maximum Workload,Average Waiting Time,Average Service Time");
        }

        // Write the data row
        printWriter.printf("%d,%d,%d,%d,%.2f,%.2f%n", concurrentUsers, totalWaitingTime,
            totalServiceTime, maxWorkload, avgWaitingTime, avgServiceTime);

    } catch (IOException e) {
        System.err.println("Error writing to CSV file: " + e.getMessage());
    }
}
```

Figure 4. Codes that output the output variables to a CSV file.

However, this will only generate the results of one simulation. I used the bash command to keep generating different stats. Save the command in Figure 3 and ‘chmod’ the bash file to give execution access, and then run the script to keep generating the simulation results to the CSV file. There are 15 seconds of sleeping time for the simulation to run. After every 100 times of simulation running, I changed the seed value so that we could get five different sets of results with different random seed values of 10, 20, 30, 50, and 12345.

Then, 15 documents, each containing 100 lines of data, were collected in the CSV file. Then convert the CSV file into an Excel file using Excel’s save as function to make the stats more convenient to analyze and make the table/figures.
Additionally, the script will also generate 15 files with 10,000 lines of data each to make the observation more universal and accurate.

After that, order all the content in the table based on the concurrent users from smallest to largest. Processing this step makes the corresponding relationships more obvious and convenient for generating tables and figures.
Finally, we collected all the required data in a well-organized table format. Then we can start to analyze the stats.

5.0 Analysis

The simulation was run for five seeds: 10, 20, 30, 50, 12345. The first part contains all the data obtained and calculated from runs with 100 lines of data. This will be the central part of the analysis.

The second part is for the 10,000 lines of data to show the system's stats more accurately. However, running all five random seeds simulations costs too much time, so only the 10,000 lines of data with a seed value of 10 are provided here.

Since we used two baselines of different priority methods to implement the simulation system, we output each result into two separate CSV files and get the difference between those two methods with the same concurrent users. This part is shown in part 3.

5.1 100 Lines of Data Analysis

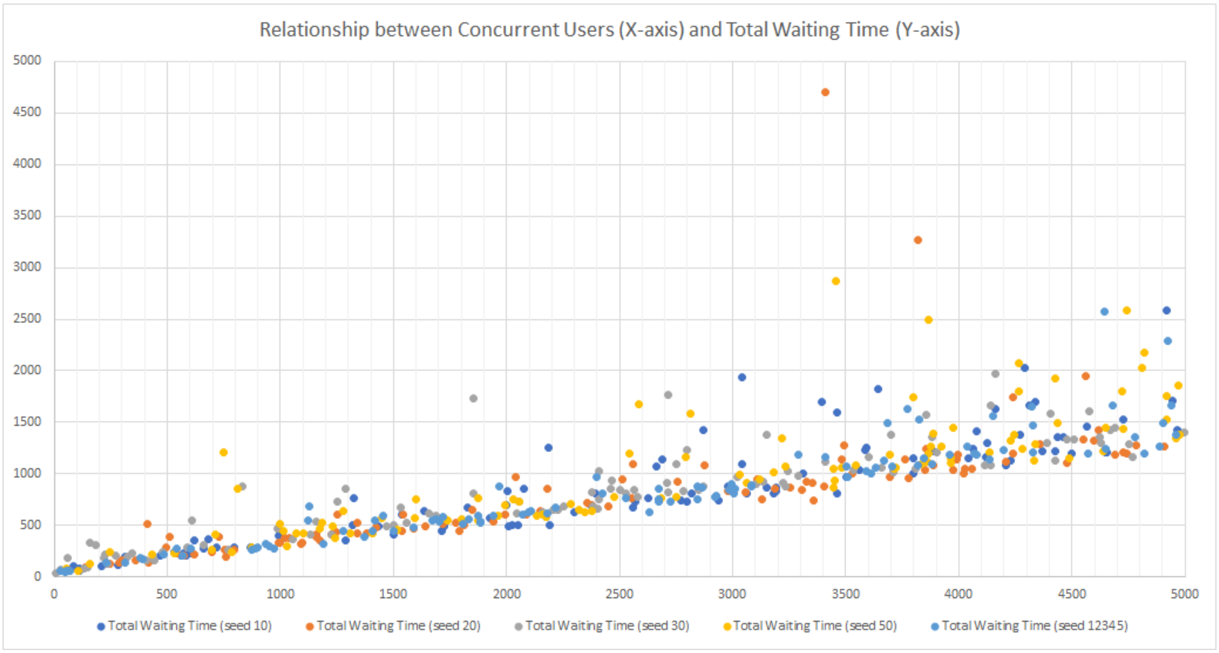


Figure 5. Relationship between Concurrent Users and Total Waiting Time

Figure 5 shows approximately a linear relationship between the concurrent users and the total waiting time. However, much outliner data is mainly above the linear line. Therefore, the different values of random seeds did not affect the total waiting time.

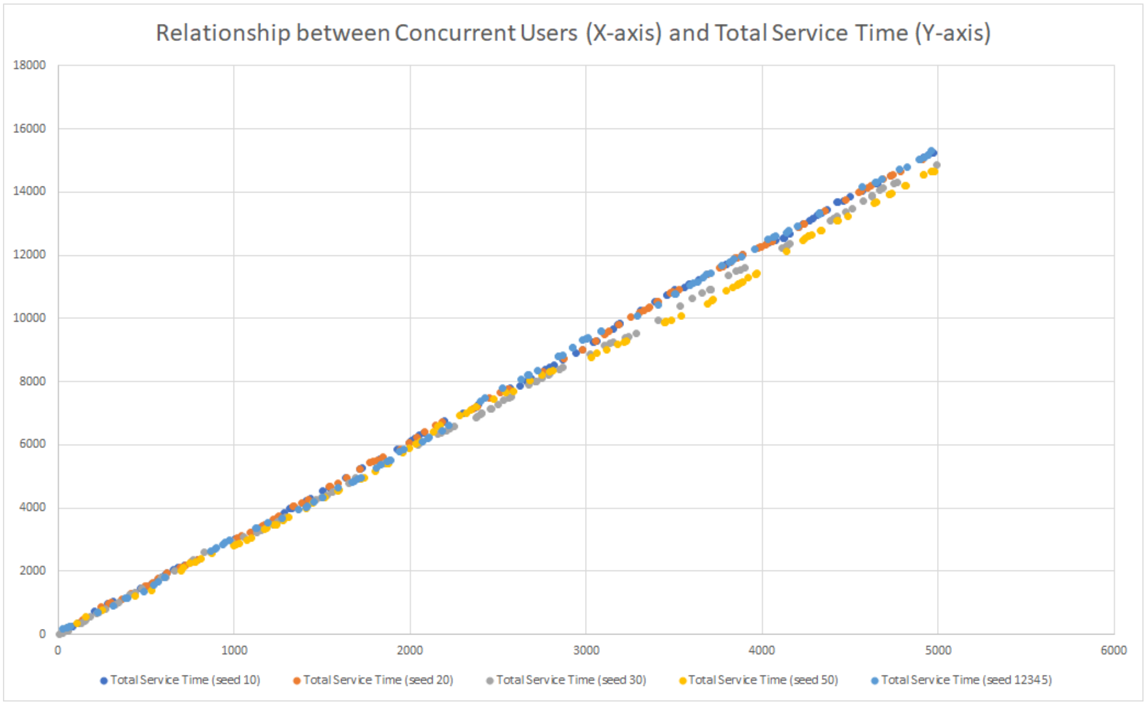


Figure 6. Relationship between Concurrent Users and Total Service Time

Figure 6 shows the relationship between concurrent users and total service time. The figure shows a perfect linear relationship. However, the different values of random seeds have little effect on the lines—for example, seed values of 10, 20, and 12345 overlap. However, the seed value of 30 and 50 shows another pattern with a lower y value generally.

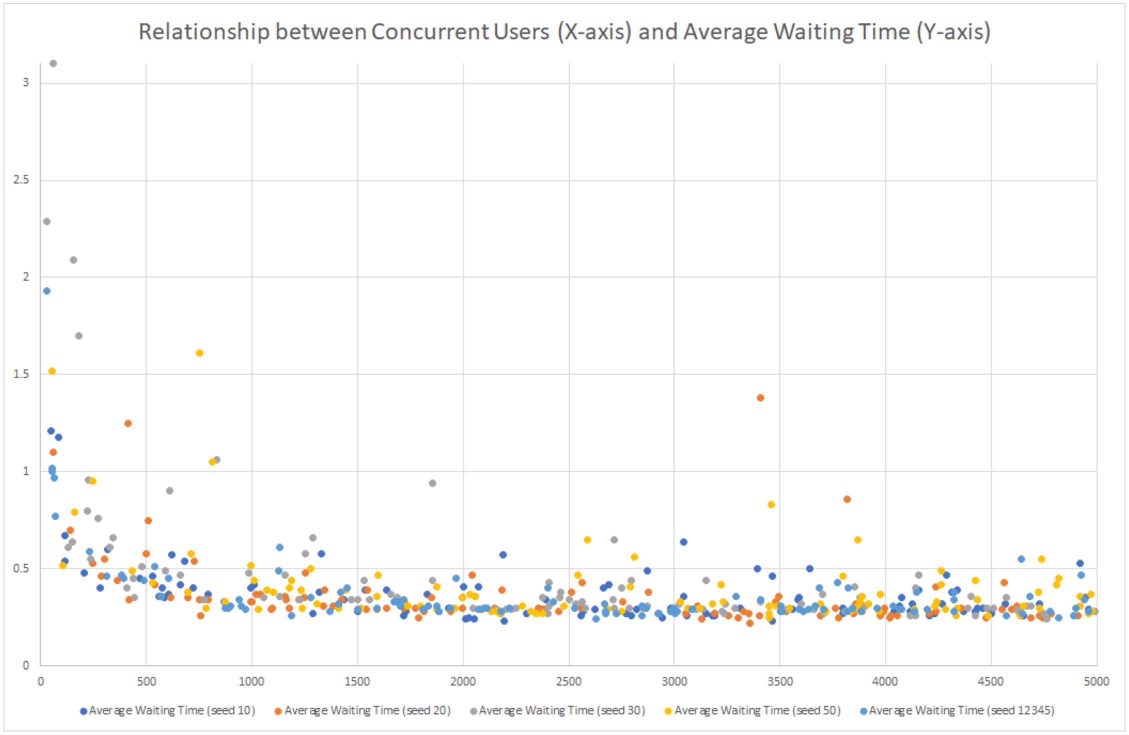


Figure 7. Relationship between Concurrent Users and Average Waiting Time

The relationship in Figure 7 between concurrent users and average waiting time shows an exponential distribution.

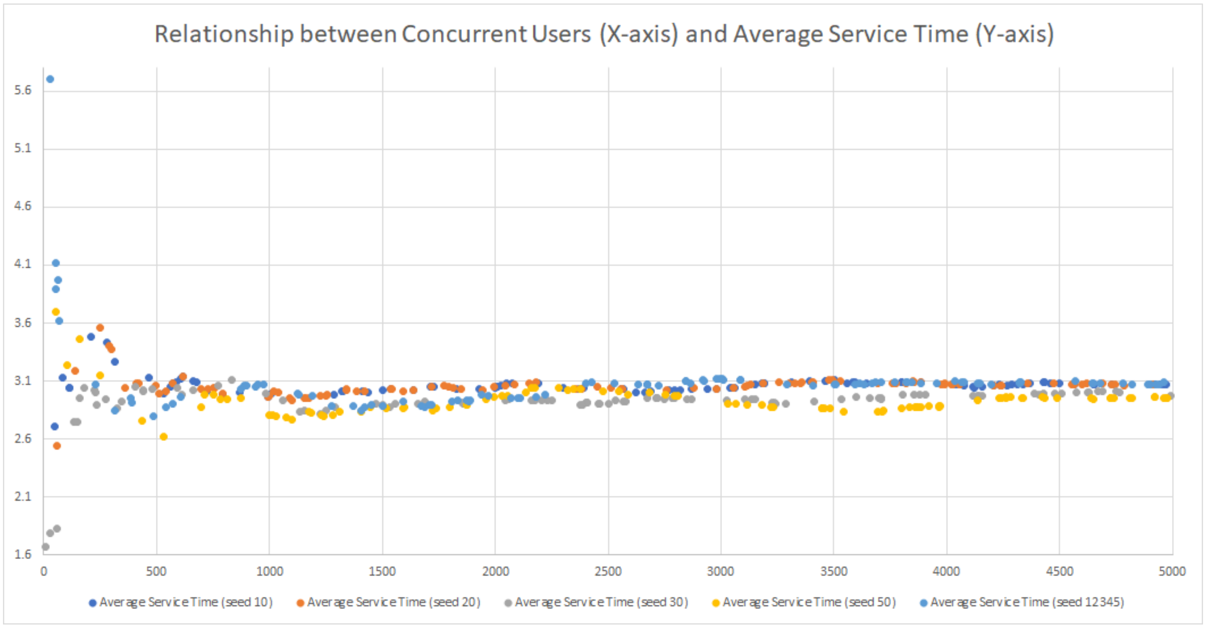


Figure 8. Relationship between Concurrent Users and Average Service Time

The relationship in Figure 8 between concurrent users and average service time also shows an exponential distribution similar to average waiting time but with more deviation.

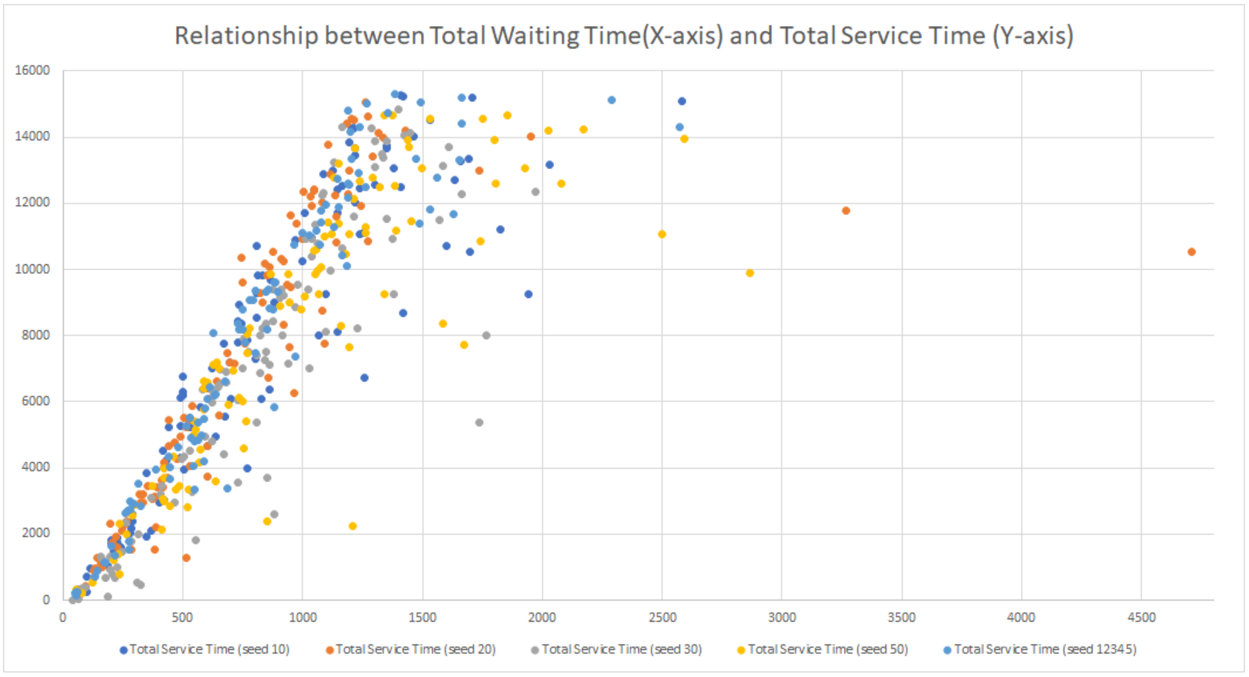


Figure 9. Relationship between Total Waiting Time and Total Service Time

The relationship in Figure 9 between total waiting time and total service time shows a linear relationship, but with some deviation on the right side of the linear line with some larger x values.

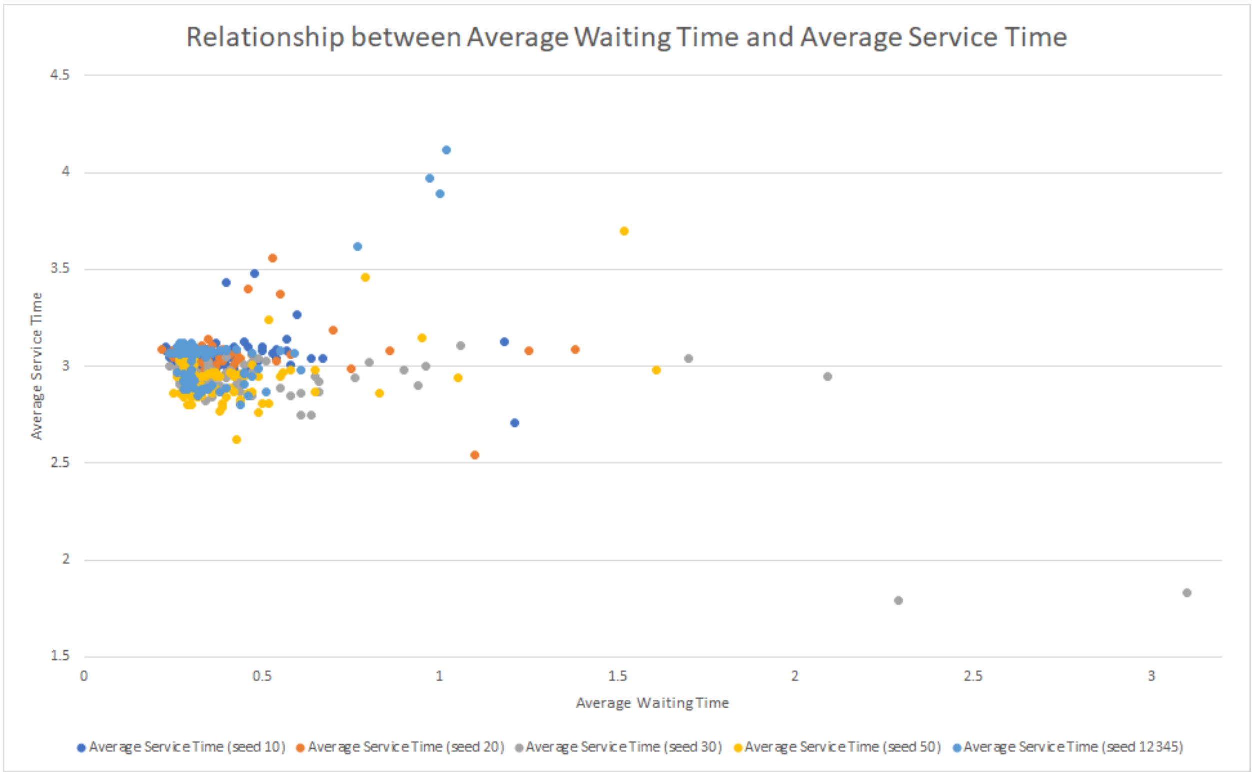


Figure 10. Relationship between Average Waiting Time and Average Service Time

In Figure 10, the average waiting time and service time ratio are 0.33 and 3, which fits the earlier parameters.

5.2 10,000 Lines of Data Analysis

The below figures are all collected and analyzed from seed value 10 with 10,000 lines of data. Since each run for a seed will cost more than two days, we only analyze one seed value as a reference.

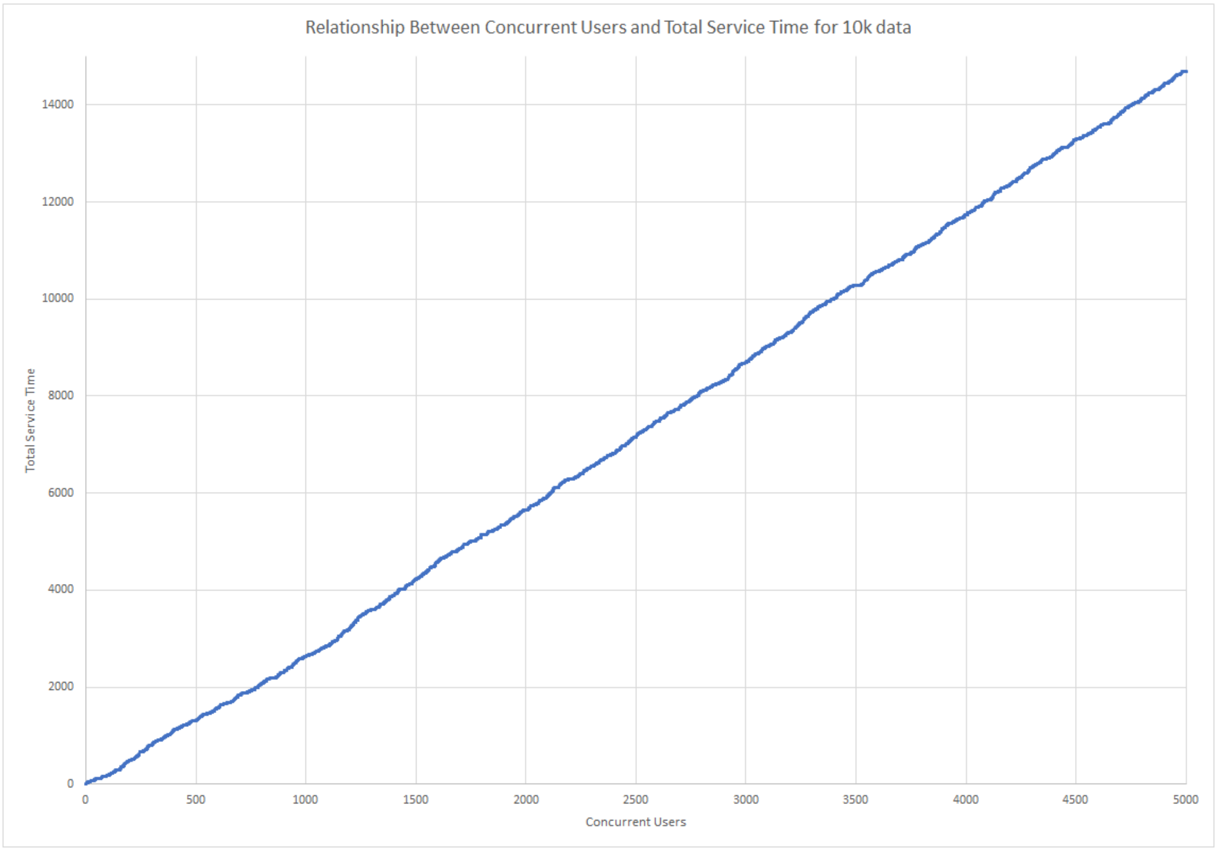


Figure 11. Relationship between Concurrent Users and Total Service Time for 10k data

For 10k data in Figure 11, the relationship between concurrent users and total service time is still perfectly linear.

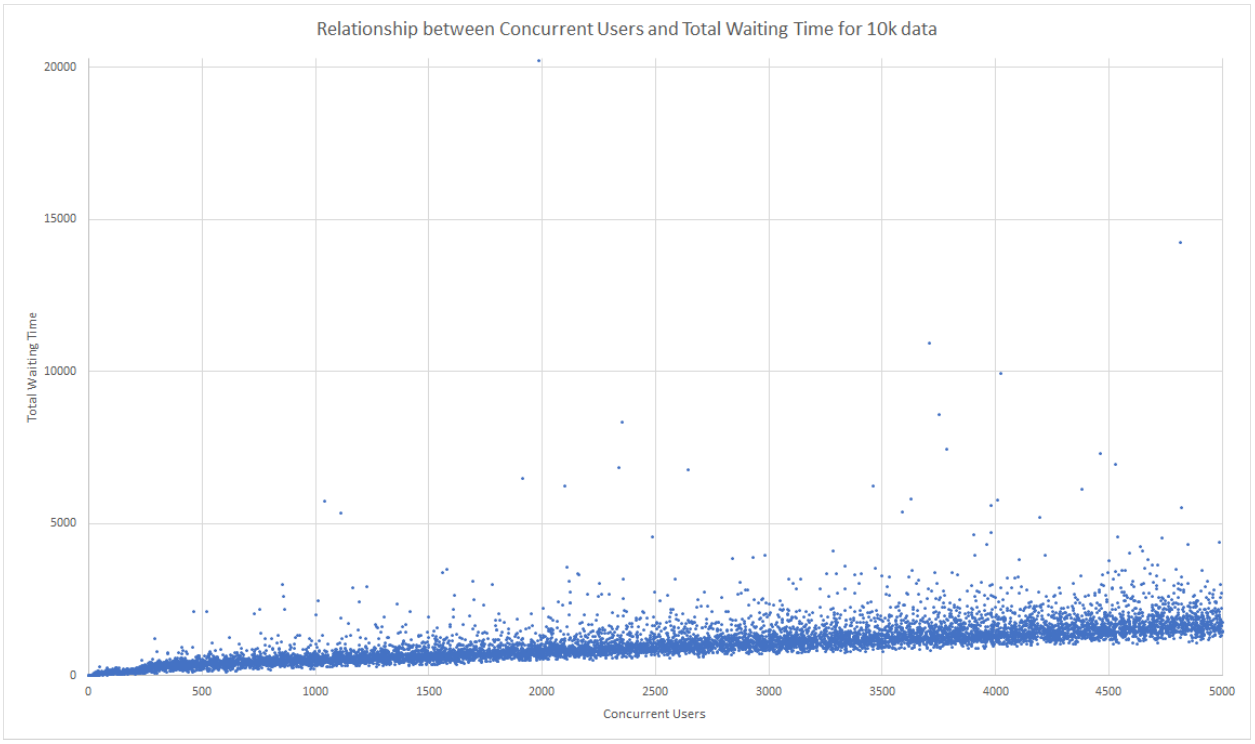


Figure 12. Relationship between Concurrent Users and Total Waiting Time for 10k Data

Like the 100 data, the 10k data in Figure 12 also fits the predicted behavior as 100.

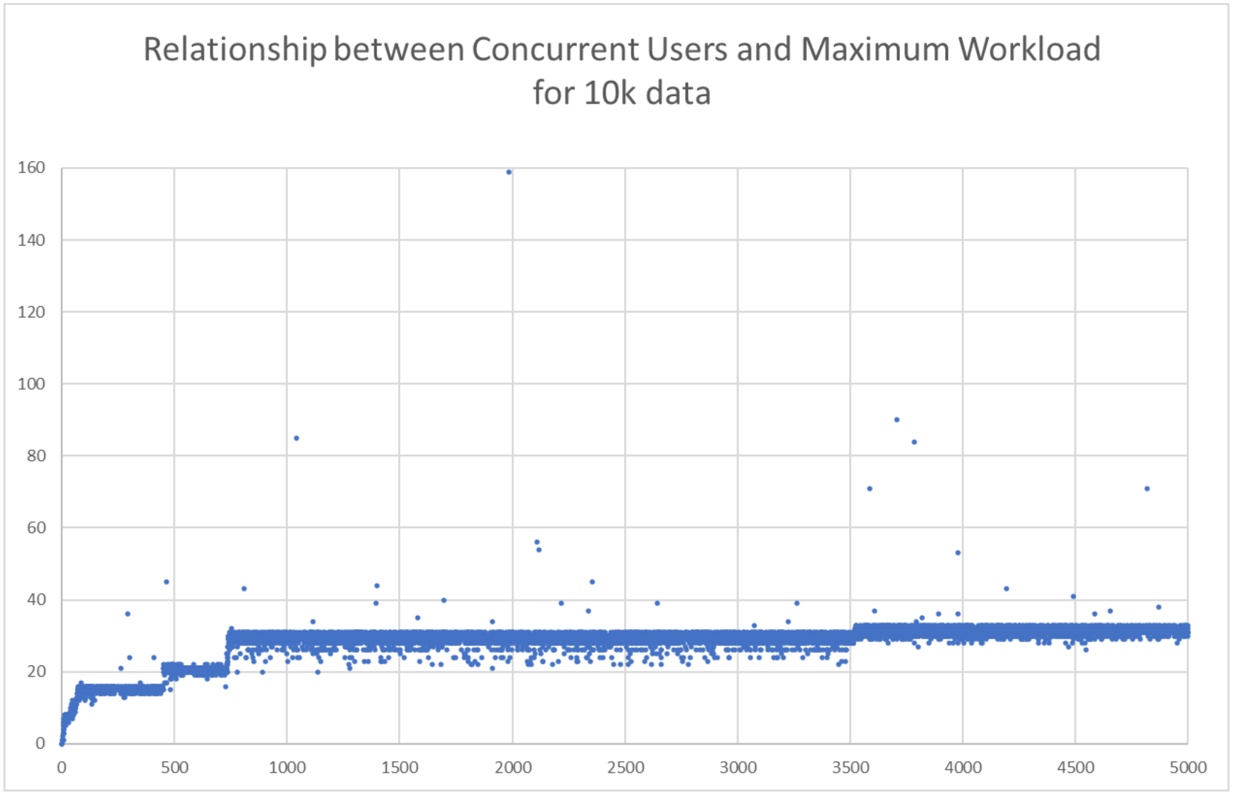


Figure 13. Relationship between the Concurrent Users and the Maximum Workload for 10k data

For 10k data in Figure 13, the relationship between users and maximum workload shows a stepped relationship with 480, 750, and 3500 users at each step.

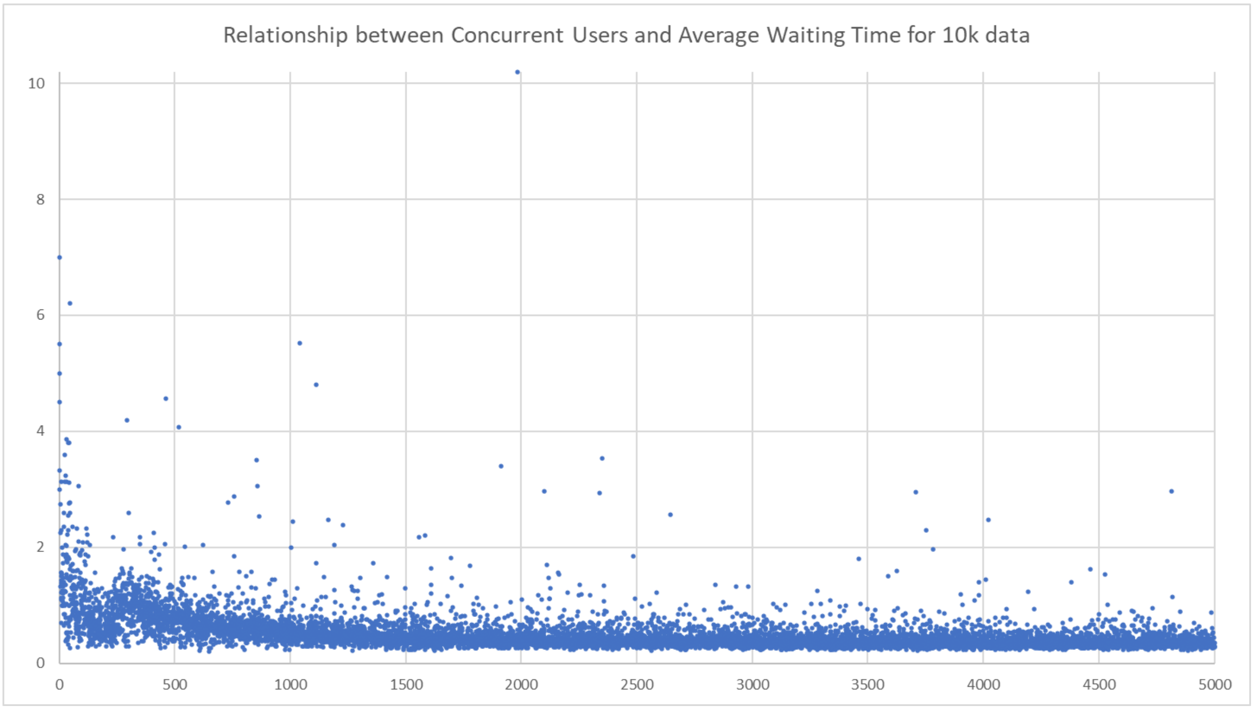


Figure 14. Relationship between the Concurrent Users and Average Waiting Time for 10k data

Figure 14 shows an exponentially distributed relationship, but the lower areas are filled in.

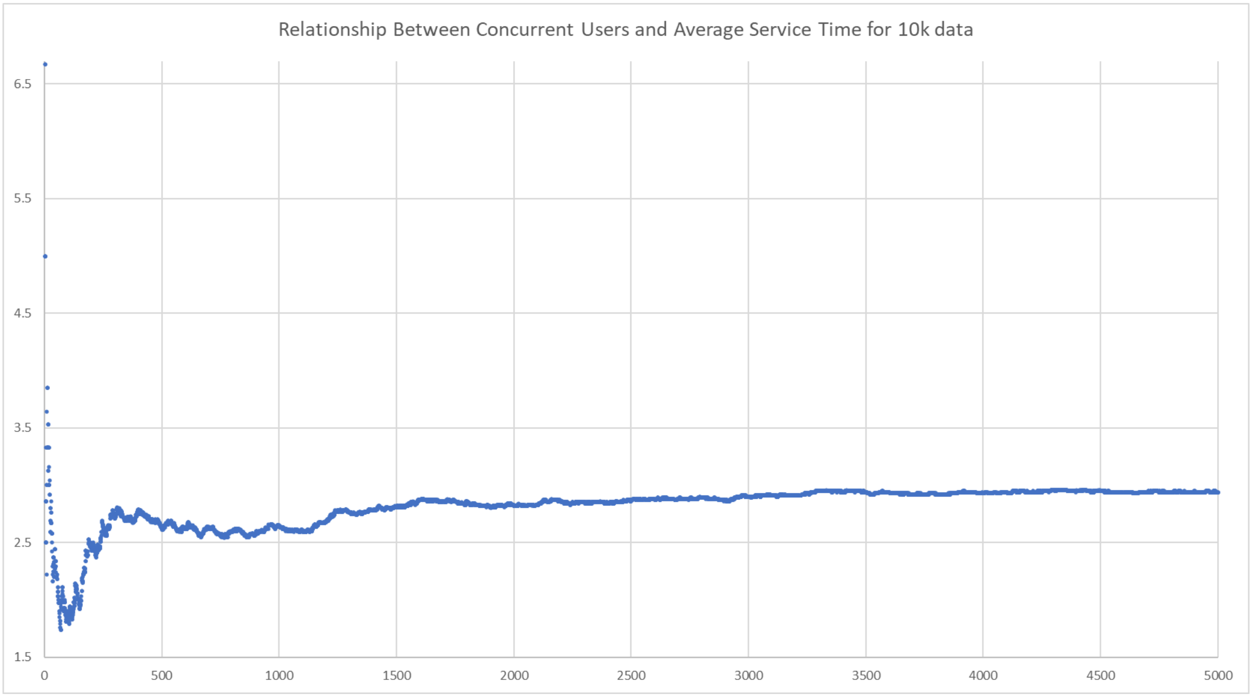


Figure 15. Relationship between the Concurrent Users and the Average Service Time for 10k data

Figure 15 shows a down slope before 100 users, then going up until about 350, and then trying to maintain a line at y = 3.

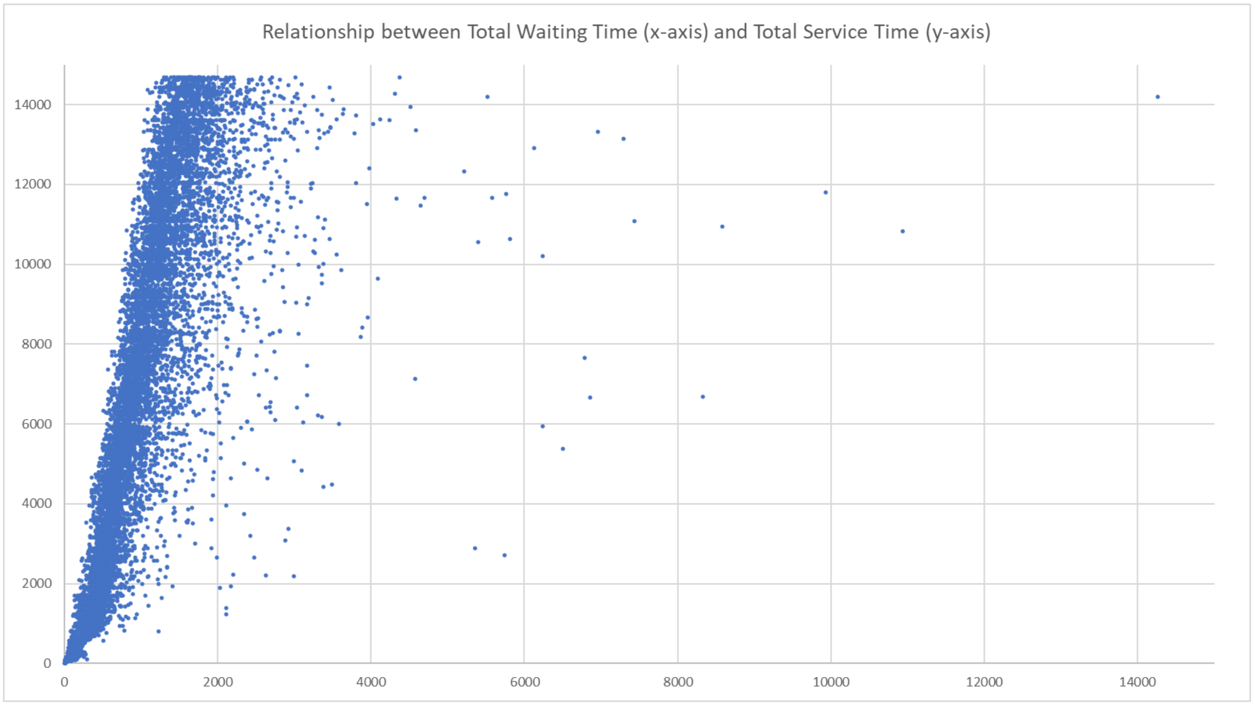


Figure 16. Relationship between the Total Waiting Time and the Total Service Time for 10k data

The 10k data in Figure 16 fits the prediction with 100 data results.

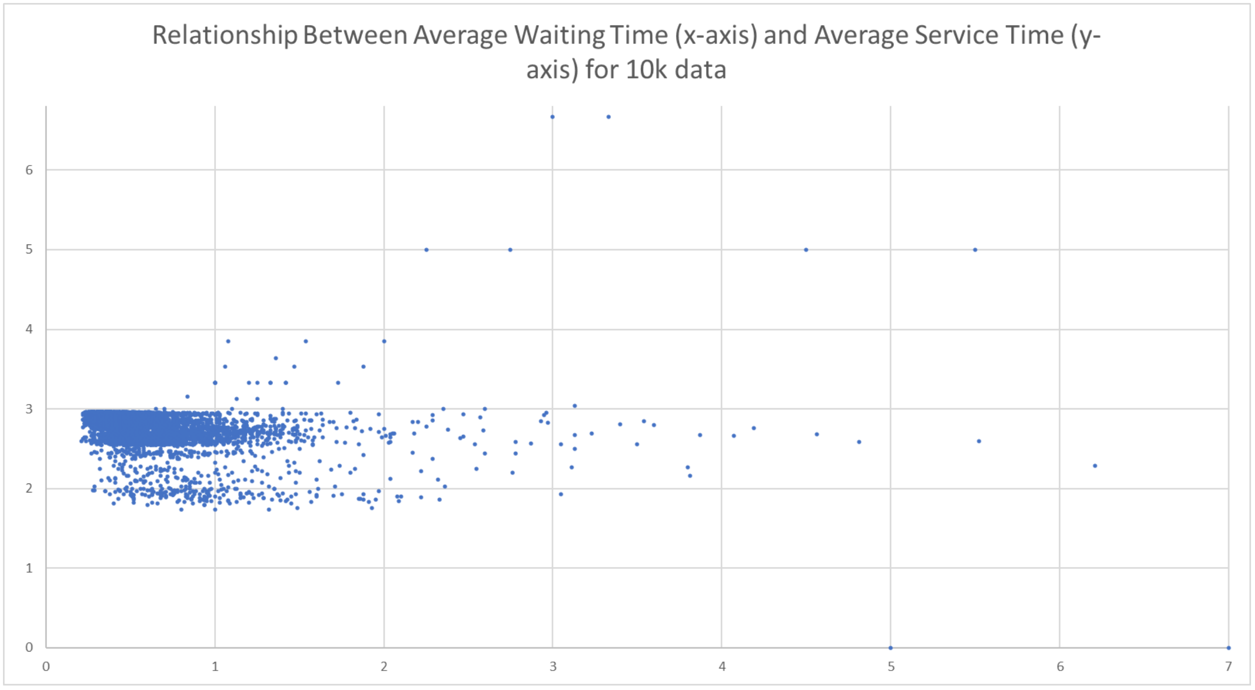


Figure 17. Relationship between the Average Waiting Time and the Average Service Time for 10k data

The 10k data in Figure 17 fits the prediction with 100 data results.

5.3 Difference of Priority Method Analysis

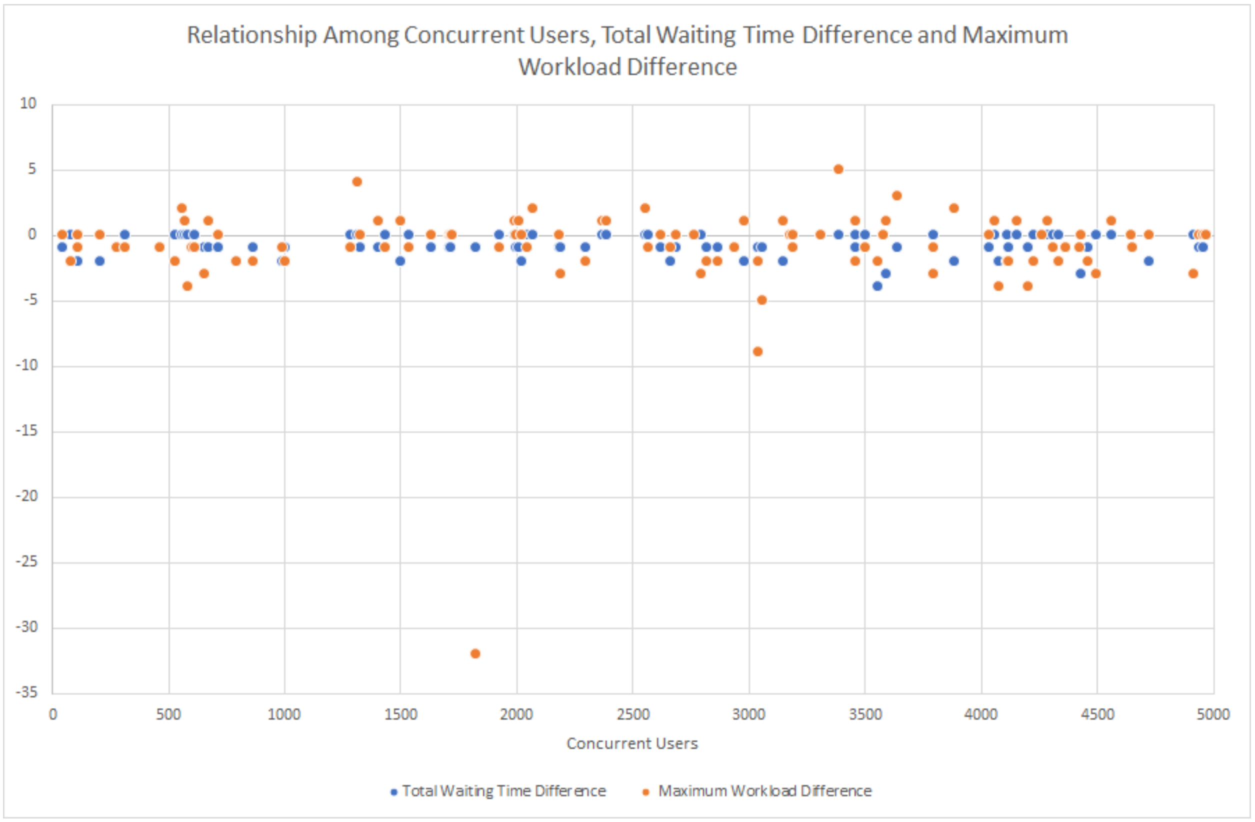


Figure 18. Relationship Among Concurrent Users, Total Waiting Time Difference, and Maximum Workload Difference

The relationship in Figure 18 among these values shows a steady line of approximately y=0 but with many variations. The difference in concurrent users and total waiting time between these two methods is insignificant. The difference in workload exists but is not significant.

6.0 Discussion and Conclusion

6.1 Conclusions

As a starter, we made some assumptions in our proposal for constructing the algorithm.
1: We assume that the number of expected visits per minute and visit duration generation follow the Poisson arrival process with mean arrival rate and lambda;
2: We assume every customer has equal priority.
3: We assume the time taken to process each request is deterministic and constant.
4: We assume that the website content is static and does not change during the simulation.

5: We assume that all the clients are accessing; the website with similar characteristics, like device types, internet-connected speed, and web browser.
6: We assume the server load is uniformly distributed across all servers.

We simulated a completed “concurrent user problem” process on our servers. We want to display the whole process and the result. However, it was too long, so we can only post some representative samples.
In order to exclude the effect caused by some particular random number, the simulation was run for five different seeds: 10, 20, 30, 50, 12345.
After that, based on the results shown in the Microsoft Excel file, we drew the graph for the relationship among various factors from our result. According to the analysis, we could find the number of concurrent users and other parameters(such as arrival time, service time, and so on)are not independent.

Conclusion 1: The relationship between the number of concurrent users and the total waiting time is nearly linear but not strictly linear.

Firstly, we found a linear relationship between the concurrent users and the total waiting time.
The formula of this relationship:
Y=1/3X
Y denoted the total waiting time(unit: ms), and X denoted the number of concurrent users.
For example, the total waiting time would be 1000 ms while there are 3000 concurrent users.
When no users existed, the total waiting time defaulted to 0.
However, for small samples(100 lines), much outliner data was found mainly above the linear line. For example, we could see that when there were 3400 users concurrently, there were two exaggerated outliers, one of these outliers was up to 2800 ms, and the other was more than 4500 ms. This type of very extreme value is beyond our expectations, and it is likely caused by the delay of the CPU cache when running the code, which has nothing to do with our model.
The dot plot diverges as the X value rises. When the number of people exceeds 3500, the time gradually begins to diverge, but the average value of divergent points still conforms to this formula. Thus the correlated value between them was less than 1. In other words, the relationship is linear but not strictly linear.

For large samples (10000 lines), its real distribution becomes clearer, which is still linear but diverges upward in the Y-axis direction, looking like countless floating bubbles.
The formula of this relationship:
Y>=1/3X
Y denoted the total waiting time(unit: ms), and X denoted the number of concurrent users.
For example, the total waiting time would be 1000 ms while there are at least 3000 concurrent users.
When there were not any users existing, the total waiting time defaulted as 0.

Additionally, based on our result, the different values of random seed did not affect the total waiting time obviously. Each seed had its extreme outliers. All seeds started to diverge while there were more than 4000 concurrent users. Therefore, the choice of a random seed is nearly independent of this relationship. Extreme values and divergence mean that the performance of the server needs to be further improved to avoid problems in practical applications, such as users being unable to log in or web pages being stuck. At the same time, it is also necessary to eliminate third-party interference as much as possible, such as repairing weak network connections and freeing excessive cache memory to avoid unstable running speed.

Conclusion 2: The relationship between concurrent users and total service time is linear.

Firstly, for both 100 lines(small sample)and 10000 lines of data(large sample),we found a linear relationship between the concurrent users and the total service time. Their formula for this relationship was nearly
Y=3X
Y denoted the total service time(unit: ms), and X denoted the number of concurrent users.
For example, the total service time would be 12000 ms while there are 4000 concurrent users.

However, the different values of random seeds have little effect on the lines. For example, seed values of 10, 20, and 12345 are almost overlapping. Moreover, the seed value of 30 and 50 shows another pattern with a lower y value generally. Nevertheless, the distribution generated from each seed is nearby. Thus we can conclude that the choice of random seed will slightly affect the results, but not decisively.
A perfect linear distribution is very beneficial in practice, meaning there are few highly long periods as the number of simultaneous clients increases. The total service time is predictable, which allows companies or schools to predict the required total service time in advance while estimating the approximate range of the total number of online users to make the website run more smoothly. Getting perfect linear relationship results also depends on a good network and device environment; that is, there will be no abnormal service time due to unforeseen their-party interference.

Conclusion 3: The relationship between the number of concurrent users and average waiting time shows an exponential distribution.

After finding the relationship between the number of concurrent users and total waiting time, we were interested in finding the relationship between the number of concurrent users and average waiting time. For a small sample (100 lines), we found that this relationship showed an exponential distribution rather than linear distribution. Of course, the average time should be zero with no users. When the number of concurrent users was in the range[1,500], we could see that the average waiting time decreased significantly as the number of people increased. When concurrent users exceed 500, the average waiting time does not change significantly, and the overall maintenance is between 2.6 and 3.1 ms. Additionally, in this model, the choice of random seed did not significantly impact the results.

At the same time, when the number of concurrent users was less than 1000, there were many extreme values. When the number of concurrent users was higher than 1000, the extreme values still existed but decreased significantly. Thus, this model is suitable for large servers with many users, and some other algorithms are needed to prevent users from facing extreme waiting times.

For large samples (10000 lines), it shows an exponentially distributed relationship, but the lower areas are filled in. During the interval [0,1500], there are plenty of outliers distributed irregularly. When there are more than 1500 concurrent users, outliers decrease significantly. In other words, as the total number of concurrent users increases, the average waiting time becomes more predictable.

Conclusion 4: The relationship between the number of concurrent users and the average service time obeys the following distribution:

Y=3.5ms, when there are few users (such as less than 10)
Y <3.5ms and Y is rapidly decreasing but not linear, when 10<=X<=100
Y=1.75ms, when X=100
Y>1.75ms and Y is rapidly increasing but not linear, when 100<X<=350
Y=2.8ms, when X=350
Y >2.5ms and Y is irregular, when 350<X<1500
Y=3ms(nearly), and Y is stable when X>=1500

Where Y denoted the average service time(unit: ms), and X denoted the number of concurrent users.
From this distribution, we can know that since it takes a certain amount of time to initialize various parameters, when the total number of concurrent users is very small, the service time equally divided among each user is relatively long. As the total concurrent users increase, the little time required for initialization is negligible when amortized. Secondly, when the total number of users is close to 100, the server achieves the best average service time. When the total number of people is between 100 to 1500, the server's average service time is unstable. When the total number of people exceeds 1500, it can be predicted that the approximate average service time is three milliseconds, so this model is suitable for large-scale network servers.

Conclusion 5: The relationship between total waiting time and total service time is linear, but with some deviation on the right side of the linear line with larger x values.

Firstly, we found a linear relationship between the concurrent users and the total service time for both small samples (100 lines) and large samples (10000 lines). Their formula for this relationship was nearly
$Y=10X$
where Y denoted the total service time(unit: ms), and X denoted the total waiting time(unit: ms).For example, when the total waiting time of the server was 1000ms, the expected total service time would be 10000ms.
The total waiting time and the total service time showed a linear relationship, indicating that the server could maintain stable operation, and no long-period waiting existed but any service. However, the dot plot showed apparent divergence when the total waiting time was higher than 1500ms. In some cases, the total waiting time was prolonged, but the total service time did not change significantly.

Conclusion 6: The average waiting time and service time ratio are 0.33 and 3, which fits the earlier parameters.

After exploring the relationship between total waiting time and total service time, the next step is to explore the relationship between average waiting time and average service time. In sample samples(100 lines), most of the points in the model are clustered in clusters between 0.25 and 0.5 milliseconds for the average wait time (x value) and between 2.75 and 3.25 for the average service time (y value). In large samples(10000 lines), most of the points in the model are clustered in clusters between 0.25 and 1 milliseconds for the average wait time (x value) and between 2.5 and 3 for the average service time (y value). There are also some scattered outliers outside the gathering range as well.
The extreme outlier value here is probably due to network quality problems. As mentioned earlier, the interference of other factors causes abnormal waiting times for some services. If this model is used in reality, various optimization measures must be taken to avoid extreme outliers in the relationship between average waiting and service times. Otherwise, it may cause damage, delay, or loss of transmitted files.
Even so, in general, the position of most points is in line with our expectations, so most users have obtained regular service progress.

Conclusion 7: The relationship between users and maximum workload shows a stepped relationship with 480, 750, and 3500 users at each step.

The relationship formula:
$Y=0$, when $X=0$
$Y<=15$, and Y is rapidly increased but not linear when $0<X<=100$
$Y=15$, when $100<X<=480$

Y=20, when 480<X<=750
Y=30, when 750<X<=3500
Y=33, when X>3500
Y denotes the maximum workload, and X denotes the number of concurrent users.

Therefore, the maximum workload of this model is predictable. We can use this model for server maintenance when essential. We need to keep the number of concurrent users within the capacity of the servers.

Conclusion 8: The difference between the two baselines, different and equal priority methods, is insignificant.

In our group proposal, we indicate two different servers:
1: Different priority servers would have different priorities, and a server with higher priority will be more likely to serve a customer.
2: Equal priority servers would have equal priorities, and each server would have the same likelihood of serving a customer.

After comparing Concurrent Users, Total Waiting Time Difference, and Maximum Workload Difference between these two methods, we found that the difference was insignificant.
First, no significant difference was found regarding the number of simultaneous online users that the two methods can carry.
Secondly, the difference between the two methods is probably published between 0 and minus 0.5 milliseconds for the total waiting time difference. Most of the points are precisely at 0 (no difference), so we conclude that the two methods' results are close but not identical regarding total waiting time.
Finally, the difference in workload exists but is insignificant. Most workload scatter points lie in the range[-5,5]. Some of these points have a positive value, and some have a negative value. The absolute average value of these points is not zero but not large.
Therefore, some of the assumptions mentioned in our proposal have been confirmed. The two operating modes are similar regarding service efficiency. Companies, schools, or other institutions can decide which method to use to run network services according to their needs, and other factors (such as economic costs) build these two methods.

6.2 Suggestions

In this simulation, we assume that the status of all clients or users is always in an ideal state by default. However, this state is impossible in reality. To apply this model in practice, you must let the server perform special processing on some client requests. For example, when the client suddenly disconnects from the network, the server should process the next client first instead of waiting indefinitely for the response of the current client. Otherwise, the service time of the user will become almost infinite until it reconnects. Additionally, to avoid DDoS attacks and prevent malicious script operations, we need to identify and not serve "too fast" requests, that is, operations that are sent far faster than human response speeds. For example, the client sends ten messages per second to different people. Finally, we need to identify "meaninglessly repetitive" requests. For example, when a customer clicks on the same link multiple times, which can be considered as Ajax / asynchronous) we should only serve one of them(processed as one request on the local browser through the front-end code) to avoid repeatedly performing the same operation and causing a waste of service time.

Secondly, for the server side, there is no perfect server in the world that can ensure 100% acceptance of all complete requests. Sometimes, the received data packets sent by the client may be damaged due to network problems, memory overflow, or other reasons. If the client's requested information is damaged, the corresponding service to the target customer cannot be carried out smoothly. When the above situation occurs, the server will receive a client request that cannot be analyzed due to the data package being damaged. In this case, it should issue a prompt message asking the client to resend the request instead of repeatedly trying to process a damaged request. Otherwise, it will give a huge waiting time for other users in the queue.

Finally, when applying these models in reality, when a mainly related parameter repeatedly appears as a very extreme outlier, professional technicians must analyze the cause in time and find a reasonable explanation and solution as soon as possible to ensure the stable operation of the server.