

Project
on
Client to Client Chat System

Avais Ahmad

173050043

Abhijit Patil

173050085

November 5, 2017

Contents

1	Introduction	3
2	Architecture Overview	4
2.1	Architecture Diagram	4
3	Types Of Request	5
4	Load Testing in Phase 2	7
4.1	System View	7
4.2	System Flow	7
4.3	Observations	7
4.3.1	Registration Request	7
5	Load Testing in Phase 3	10
5.1	System View	10
5.2	System Flow	10
5.3	Load Test	10
5.4	Observations	11
5.4.1	Registration Request	11
6	Approach to optimize the system	14
6.1	Previous system analysis	14
6.2	Optimization Technique	14
7	Conclusion	16

1 Introduction

Our project provides a platform for a client to client chatting in a very efficient manner by using multi-threading and socket programming. In this two clients can chat after authentication from the server. The user has to register using a user name and password. A unique port number will be assigned to it by the server which will be later used for creating a new connection for chatting. User name should be unique. After registration client has to log in. Once his user-name and password are verified by the server he can either wait for other clients to connect or he can request another client who is waiting To request another client to connect the client have to enter the user-name of the client to whom he wants to chat . The server will send the port of that client. Using this port a new socket can be created for both clients to chat. Both clients have to send message “bye” to end the chat connection.

2 Architecture Overview

2.1 Architecture Diagram

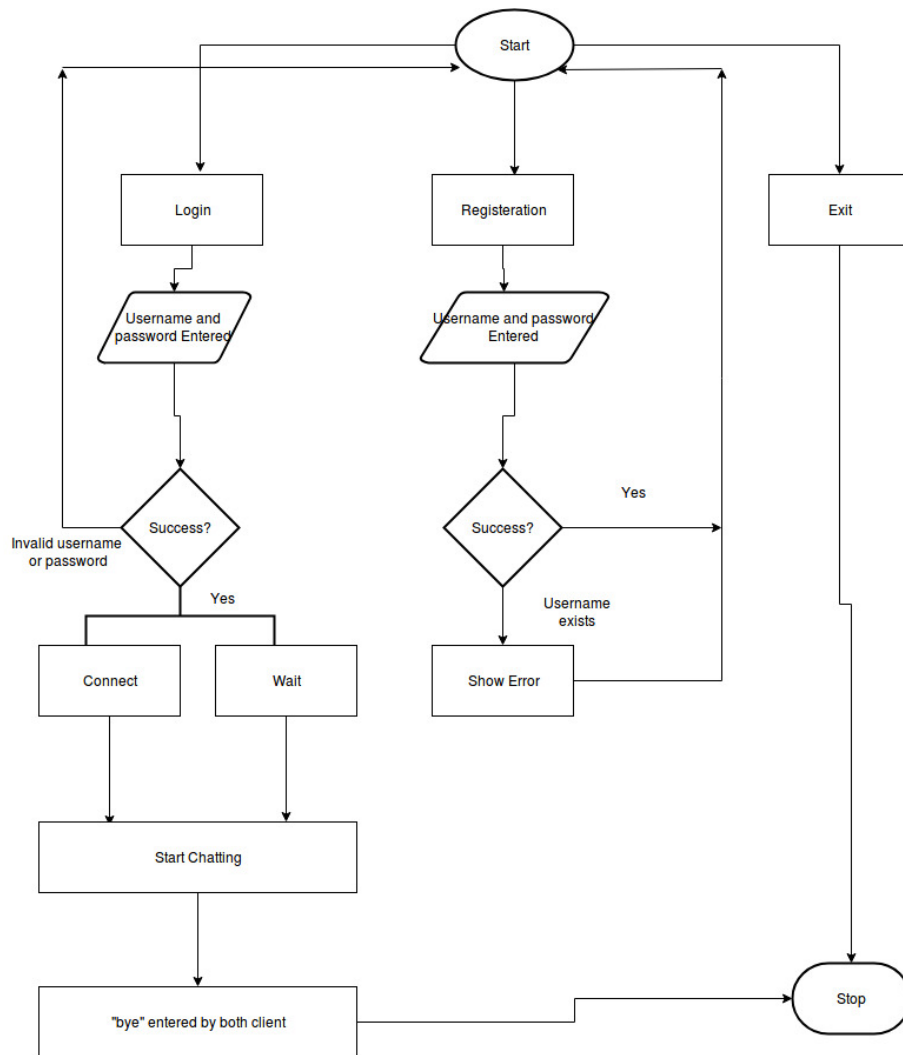
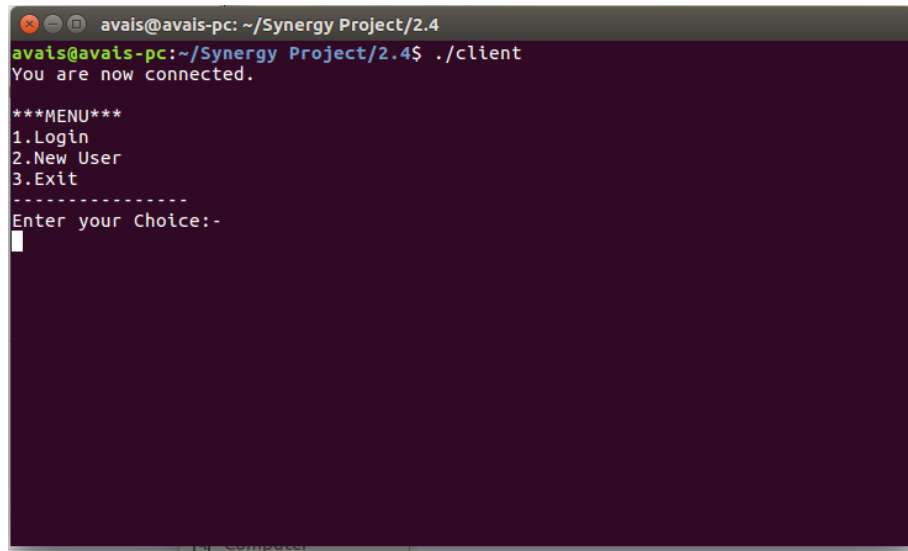


Figure 1: Architecture

3 Types Of Request

In our project, we have a different type of requests:-



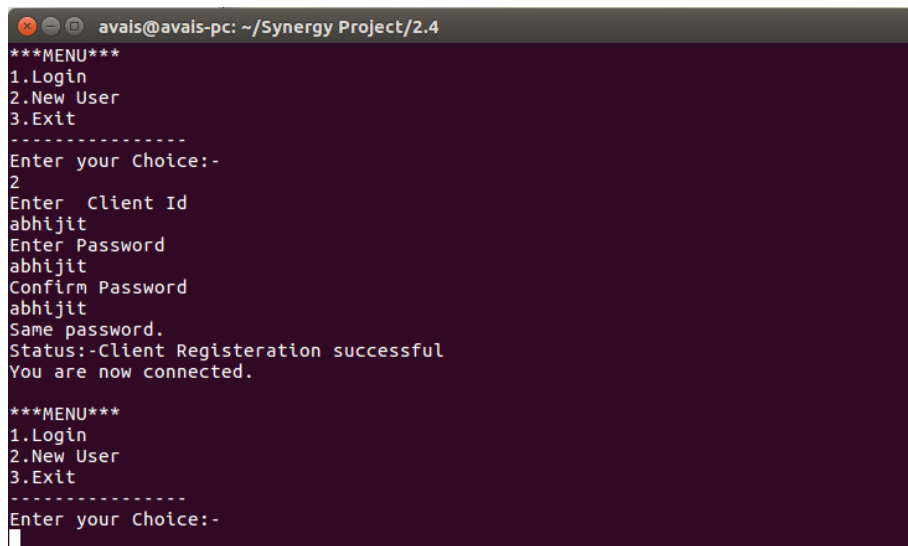
```
avaits@avaits-pc: ~/Synergy Project/2.4
avaits@avaits-pc:~/Synergy Project/2.4$ ./client
You are now connected.

***MENU***
1.Login
2.New User
3.Exit
-----
Enter your Choice:-

```

Figure 2: Menu

1. Registration:- In this request, the client establishes the connection with Server and do registration operation with user id and password. On completion of the request, Server gives response message.



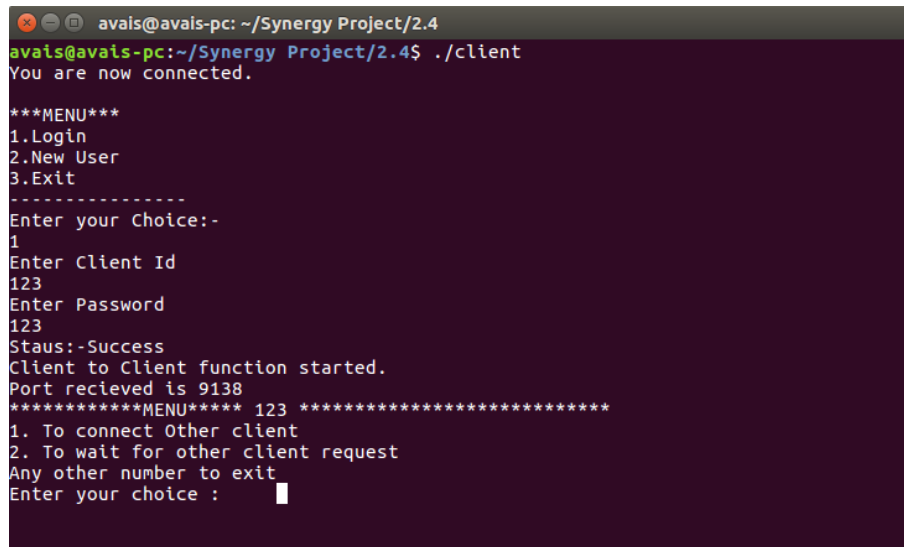
```
avaits@avaits-pc: ~/Synergy Project/2.4
***MENU***
1.Login
2.New User
3.Exit
-----
Enter your Choice:-
2
Enter Client Id
abhijit
Enter Password
abhijit
Confirm Password
abhijit
Same password.
Status:-Client Registration successful
You are now connected.

***MENU***
1.Login
2.New User
3.Exit
-----
Enter your Choice:-

```

Figure 3: Registration Request

2. In this request, client login with credentials and ask for a self-port number from the server. then client creates a socket on self-port number wait for a client request
3. In this request, client login with credentials and ask for an another client port number whom it wants to connect. then the client sends a request on that port number.

A terminal window titled 'avals@avals-pc: ~/Synergy Project/2.4' showing the execution of a client program. The user runs './client' and the program outputs 'You are now connected.' followed by a menu. The user selects '1.Login', enters '123' for both Client Id and Password, and receives 'Staus:-Success' (sic) and 'Client to Client function started.' The program then displays 'Port recieved is 9138' (sic) and a second menu. The user is currently at the 'Enter your choice :' prompt.

```
avals@avals-pc: ~/Synergy Project/2.4
avals@avals-pc:~/Synergy Project/2.4$ ./client
You are now connected.

***MENU***
1.Login
2.New User
3.Exit
-----
Enter your Choice:-
1
Enter Client Id
123
Enter Password
123
Staus:-Success
Client to Client function started.
Port recieved is 9138
*****MENU***** 123 *****
1. To connect Other client
2. To wait for other client request
Any other number to exit
Enter your choice : █
```

Figure 4: Login Request

4. After the establishment of Connection to another client, it breaks the connection with the server and starts message on their private ports.

4 Load Testing in Phase 2

In this phase, We've written a closed-loop load generator to test the system, in which there are N threads continuously sending requests to the servers without any think time in between the subsequent requests.

4.1 System View

- The load generator runs for 300 seconds and measures average response time and throughput of the system for these 300 seconds.
- The Front end and Back end servers run on one machine with multiple CPU Cores on it, while the Load Generator runs on another.
- Front End will interact with Back end server which is MYSQL database
- We have implemented Multi-Threaded server and Load Generator.

4.2 System Flow

- The main thread of the load generator spawns N threads.
- Each thread runs in loop for 300 seconds. each thread creates client socket. After completion of First request it generates new request.
- For each of these request, we maintain a TCP socket connection to the Front end Server, which in turn opens MYSQL connections to the Back end Server, one for each connection from the Load Generator.
- Requests are sent continuously from the Load Generator to the Front end Server for these N threads without any delay in between the requests, Front end Server assigns these requests to the Back end Server using the corresponding outgoing connection of the current thread.

4.3 Observations

Upon increasing the number of users to the system we calculated average response time per request . Upon observation, it is clear that Average Response Time per request increases on increasing the number of users N in the system as defined by the number of threads spawned on the load generator initially. After a certain point response time increases gradually as when the CPU hits its maximum capacity and could not perform better. Hence it takes more amount of time to service any particular request by the system.

4.3.1 Registration Request

In this Load generator, we started with low number of threads and gradually increase the number of threads till it reaches to stable value of throughput. For N=35 , our system reaches full %CPU Utilization . Our system is bottleneck by Our CPU as you can see in Figure?? where all 4 cores are reached to 100% utilization and for very large number threads MYSQL is also giving low

performance as it is unable to maintain more connection at a time. In load generator ,Thread has to wait in loop for MYSQL connection till it gets connected successfully.

1. Throughput Of the System

Observation are in Table shown below

No of Threads	Throughput
1	20.79
2	27.53
5	62.65
10	89.72
15	104.70
18	108.09
20	112.11
25	114.54
30	115.47
35	117.22
40	117.62
50	118.49
100	118.69

And the corresponding graph plotted against these values is shown above:

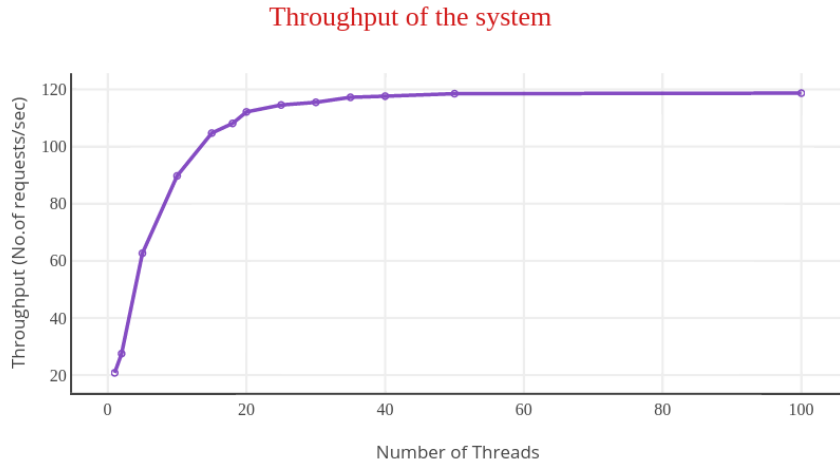


Figure 5: Throughput

2.Latency Of the System

No of Threads	Latency
1	48032
2	72585
5	79763
10	111472
15	143315
18	166693
20	178480
30	260227
35	320779
45	380646
50	423090

And the corresponding graph plotted against these values is shown above:

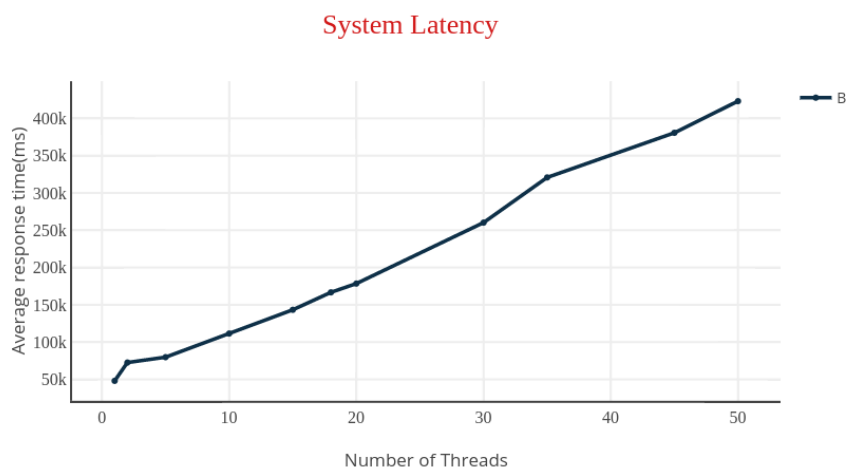


Figure 6: Latency of the System

5 Load Testing in Phase 3

In this phase, We've written a closed-loop load generator to test the system, in which there are N threads continuously sending requests to the servers without any think time in between the subsequent requests.

5.1 System View

- The load generator runs for 300 seconds and measures average response time and throughput of the system for these 300 seconds.
- The Front end and Back end servers run on one machine with multiple CPU Cores on it, while the Load Generator runs on another.
- Front End will interact with Back end server which is REDIS-SERVER
- We have implemented Multi-Threaded server and Load Generator.

5.2 System Flow

- The main thread of the load generator spawns N threads.
- Each thread runs in loop for 300 seconds. each thread creates client socket. After completion of First request it generates new request.
- For each of these request, we maintain a TCP socket connection to the Front end Server, which in turn opens Redis-server connections to backend server.
- Requests are sent continuously from the Load Generator to the Front end Server for these N threads without any delay in between the requests, Front end Server assigns these requests to the Back end Server using the corresponding outgoing connection of the current thread.

5.3 Load Test

- The main thread calls join and waits for each of the N user threads, each of which runs for 300 seconds.
- Within loop, we take the system time stamp before calling a request and after receiving a response respectively, to get the response time for that request.
- We noted all such response times to get cumulative response time for number of requests as 'R' .
- We calculate the average response time 'i' across 'r' requests

$$i = \frac{totalResponsetime(R)}{NumberOfRequests(r)} \quad (1)$$

- Now, we calculate the throughput for closed loop systems as per formula:

$$X = \frac{NumberOfRequestServed}{TotalTime} \quad (2)$$

5.4 Observations

Upon increasing the number of users to the system we calculated average response time per request . Upon observation, it is clear that Average Response Time per request increases on increasing the number of users N in the system as defined by the number of threads spawned on the load generator initially. After a certain point response time increases gradually as when the CPU hits its maximum capacity and could not perform better. Hence it takes more amount of time to service any particular request by the system.

5.4.1 Registration Request

As it is shown in Figure3 on Page5 , new user has to provide Unique Client ID and password to register into the database. Our load generator generates new client socket and bombards registration request on Server.

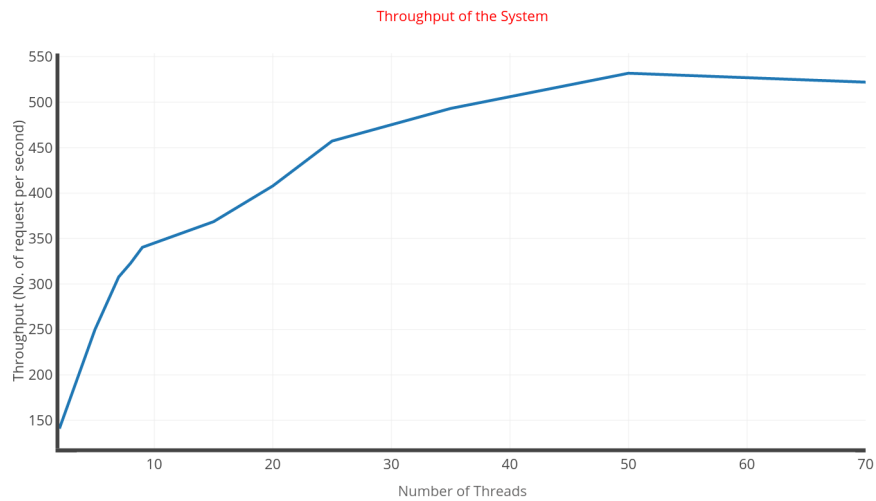
Load generator creates ' N ' threads, each thread generates a request in a loop. For each request, it creates client socket and establishes the connection with the server. then it sends "Registration" message to inform the server that client wants to perform registration operation. On receiving an appropriate response from the server, Client sends Client Id and password. After completion of a request, the new request will generate.

In Load generator, we started with the low number of threads and gradually increase the number of threads till it reaches to a stable value of throughput. For $N=50$. Our system is bottleneck by Redis-Server as for very large number threads Redis-server is also giving a low performance as it is unable to maintain more connection at a time. In load generator, Thread has to wait in a loop for Redis connection till it gets connected successfully.

1. Throughput Of the System
Observation are in Table shown below

No of Threads	Throughput
2	140.93
5	249.84
7	307.71
8	322.81
9	340.36
15	368.54
20	407.89
25	457.27
35	493.13
50	531.86
70	522.06

And the corresponding graph plotted against these values is shown above:



2.Latency Of the System

No of Threads	Latency
2	14135
5	19958
7	22683
8	24715
9	26371
15	40635
20	48974
25	54623
35	70939
50	93993
70	134114

And the corresponding graph plotted against these values is shown above:

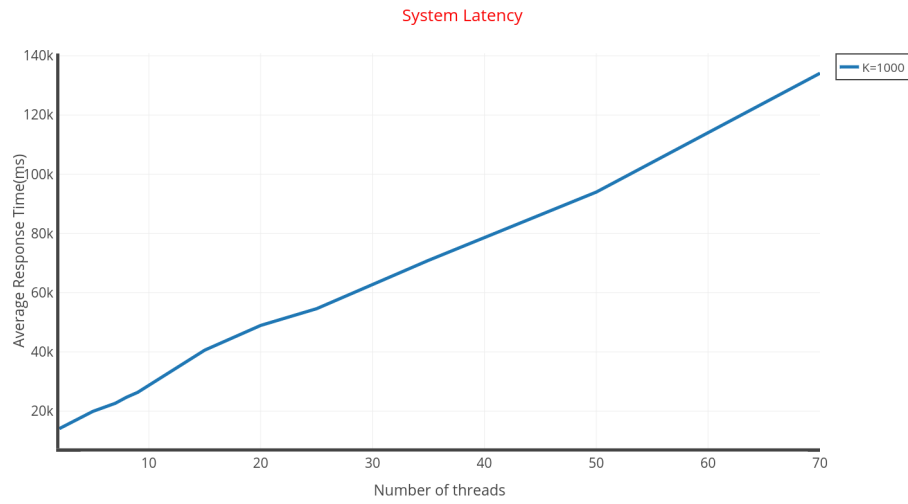


Figure 7: Latency of the System

6 Approach to optimize the system

6.1 Previous system analysis

In the previous phase, our system was bottleneck by CPU where all four cores of the system were 100% utilized. so we profiled our code with "oprofile". MYSQL DB connection was taking most of the CPU utilization. we had to build multi-threaded server. we had designed lock free system for better performance as lock serialize the performance of the system.

6.2 Optimization Technique

In this phase, we have used REDIS as our back-end server. Redis is an open-source in-memory database project implementing a networked, in-memory key-value store with optional durability. we have removed time-consuming operations like memory free functions.

after we changed REDIS as our back-end server. It optimized the System drastically but later it was utilizing the network bandwidth fully. first, we test our system by connecting client and server by LAN connection. but it was limited by network bandwidth. so just to make sure whether our system is bottleneck by Network. we tried to make the client to server connection by peer to peer ad-hoc connection(Hotspot).

we are also limited by max connection handle by REDIS server which led to abrupt termination. Due to huge number request generation, our system reaches to maximum mapping value of virtual memory which is max_map_count. This file contains the maximum number of memory map areas a process may have. Memory map areas are used as a side-effect of calling malloc, directly by mmap and mprotect, and also when loading shared libraries. which doesn't allow to create connection after certain limit. so we have to change that value. While most applications need less than a thousand maps, certain programs, particularly malloc debuggers, may consume lots of them, e.g., up to one or two maps per allocation.

Along with Network saturation, Our system saturates memory of the system . for that we have used some pthread libraries function to free memory after completion of each thread.

This are some of the reading from Client Server Connection over LAN.

No of Threads	Latency
2	152.77
3	186.74
4	191.95
5	190.97
10	190.17

And the corresponding graph plotted against these values is shown above:

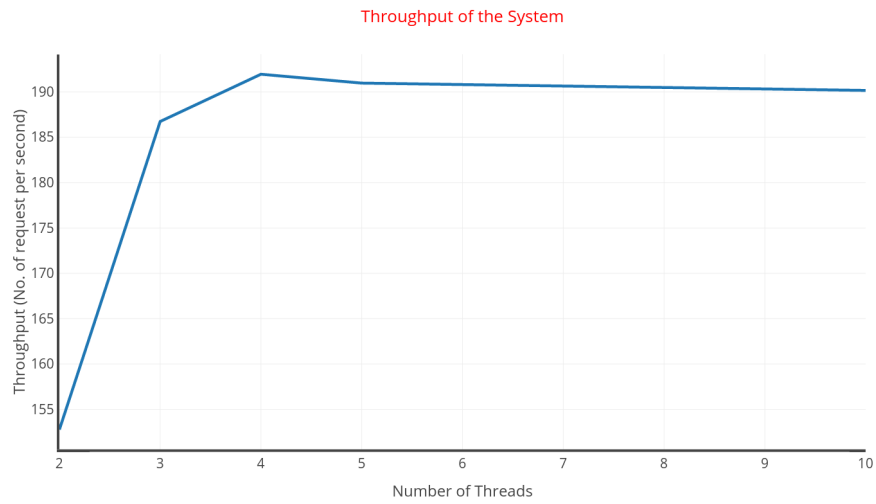


Figure 8: Latency of the System

7 Conclusion

From this project phase , we learned how to optimize our system capacity. we have learned how to handle bottleneck from MYSQL database which led to saturation of CPU. NOSQL database are faster so we used REDIS as our back-end server. which drastically optimize system performance. we have learned how to handle disk saturation by using some memory deallocation function which are less time consuming. Throughput of the system gradually increases as increase in number of threads but after some threshold value it fully utilized the system. Average latency of system keeps on increasing because of waiting time for threads increases.