File server with backend authentication

Phase 2 : Load Testing

Major Amit Pathania 163054001
Nivia Jatain 15305R007

Department of Computer Science and Engineering
Indian Institute of Technology, Bombay

October 2017

Contents

List of Figures

1 Introduction

File server with backend authentication server has been implemented using TCP sockets in C++. There are two servers: Server1 and Server2. Server1 (File server) receive TCP requests from client and requests a username and password from client for either creating new account or for requesting a file. Server1 will send this username and password to Server2 (Backend authentication server) to authenticate the user or add new user account. Once authenticated, the client can then request files from the server1 and if file is present, server1 will send the file to client. Server1 is multithreaded to handle multiple client requests. Server2 will is multiprocess to handle multiple authentication requests from Server1. The broad architecture of our system is described in figure1.

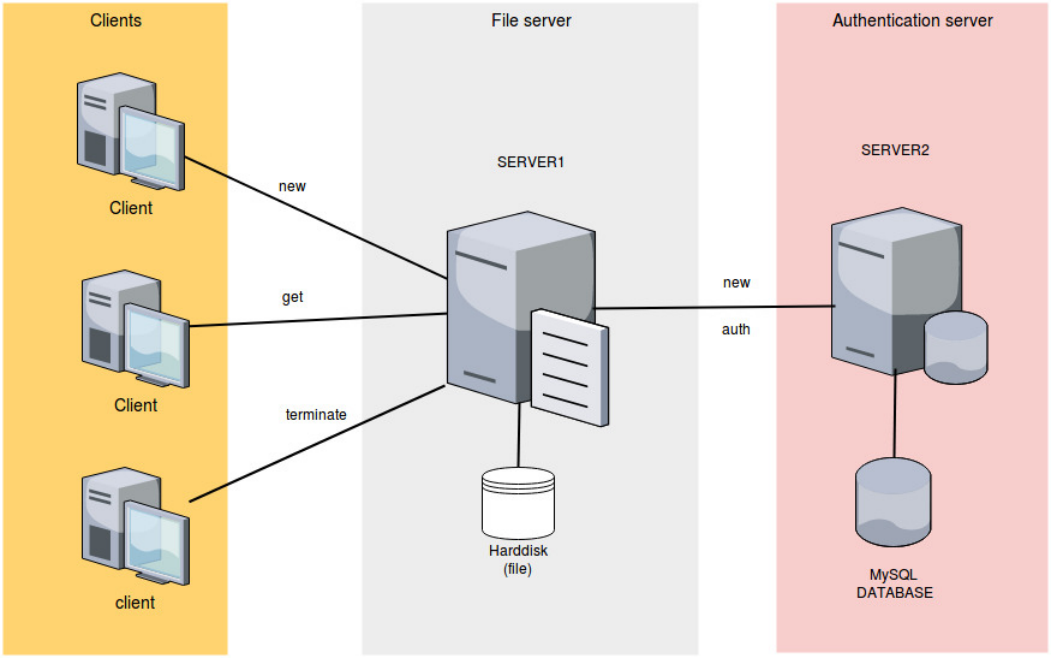

Figure 1: System Architecture

Our load generator performs close loop test using N threads.The response time, T , in a closed system is defined to be the time from when a request is submitted until it is received. In the case where the system is a single server (e.g. a web server), the server load, denoted by ρ , is defined as the fraction

of time that the server is busy, and is the product of the mean throughput X and the mean service demand (processing requirement) E[S].

2 Load generator

2.1 Design of load generator

We are performing a closed loop test where new job arrivals are only triggered by job completions.In order to simulate N concurrent users, we create N threads in our load generator. Each of the N threads emulate one user by issuing one request, waiting for it to complete, and issues the next request immediately afterwards.There is no think time between requests.

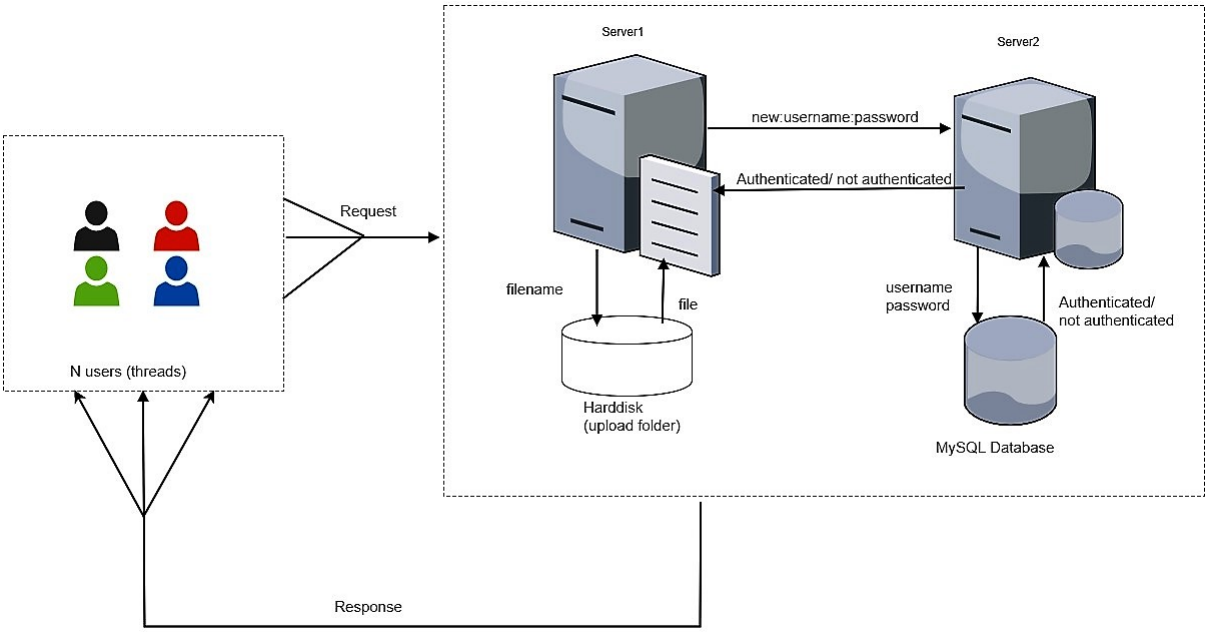

Figure 2: Design of load generator

Code snippets of the code to generate N threads where NUM_THREADS is number of threads defined by user. The threads are created using pthread_create and a pointer to a function (user_work) that the thread runs is provided in the function call :

```
for(id = 0; id < NUM_THREADS; id++ ) {
rc = pthread_create(&threads[id], NULL, user_work, (void *)id);
if (rc) {
cout << "Error:unable to create thread" << rc << endl;
exit(-1); } }
```

The user_work(void *threadid) is function which each thread runs and is used by thread to issue request, waiting for it to complete and again issuing the request.The connect_to_server (threadid) function creates a TCP connection to server, fetches the file or tries to create a new account and closes the connection to server. There are two ways to run load generator :

Approach I: Running load generator for specific time Here,user can give total duration for which timer should run and user_work(void *threadid) function keep issuing the request after completion till timer goes off. Here, however number of requests send by different users/threads within the stipulated time may not be same as we increase the number of threads because request of different users/threads may take different time for completion. Pseudo code for same:

```
void *user_work(void *threadid){
result *ret = (result *)malloc(sizeof(result));
int t_complete_counter=0;
int t_err_counter=0;
int flag=0;
while(t_duration < RUN_TIME){ // to check how much time has been elapsed
t_duration = difftime(current_time,t_start);
th_start= std::chrono::high_resolution_clock::now();
flag=connect_to_server(threadid);
th_end = std::chrono::high_resolution_clock::now();
std::chrono::duration<double> th_elapsed = th_end -th_start;
if(flag!=-1){
t_complete_counter++;
resp_time.push_back(th_elapsed.count());}
else{
t_err_counter++; }
current_time = time(NULL);}
vector<double>::iterator v = resp_time.begin();
while( v != resp_time.end())
total_resp_time=total_resp_time+ *v;
v++;
ret->total_resp_time=total_resp_time;
ret->comp=t_complete_counter;
ret->err=t_err_counter;
pthread_exit((void*)ret);}
```

Approach II: Running load generator for fixed number of requests per user Here,user can give total number of requests per user to execute and user_work(void *threadid) function of each thread keep issuing the request till per thread requests defined by user are completed.This approach ensures that load generator generates equal work load(number of requests) for each user(thread). The pseudo code for same:

```
void *user_work(void *threadid){
result *ret = (result *)malloc(sizeof(result));
int t_complete_counter=0;
int t_err_counter=0;
int num_req=0;
while(num_req < MAX_REQ){ // to check how much requests have been generated
num_request++;
cout<<t_duration<<endl;
th_start= std::chrono::high_resolution_clock::now();
flag=connect_to_server(threadid);
th_end = std::chrono::high_resolution_clock::now();
std::chrono::duration<double> th_elapsed = th_end -th_start;
if(flag!=-1){
t_complete_counter++;
resp_time.push_back(th_elapsed.count());}
else{
t_err_counter++;}}
vector<double>::iterator v = resp_time.begin();
while( v != resp_time.end())
total_resp_time=total_resp_time+ *v;
v++;
ret->total_resp_time=total_resp_time;
ret->comp=t_complete_counter;
ret->err=t_err_counter;
pthread_exit((void*)ret);}
```

We tried our experiments with both the approaches.

2.2 Type of load/requests

Load generator can generate two types of load (or requests):

- **To create new user account**. In this case, "new" keyword followed by username and password will be sent to server1 (**new: uname: upassword**). Server1 will send "new" keyword followed with username and password to Server2 . Server2 will try to create user account. If some user account with that username exists, account will not be

created else new user account will be created. Same will be informed to server1 and server1 will inform the same to client.

During load test, we generate request for creating new account. Server1 forwards request to Server2 which tries to create the new account. In this case,**there is no file transfer (disk access at Server1) and only new socket connection** is created between Client, Server 1 and server2 for each request. **Server2 will connect to mySQL database for each request** and try to create account.

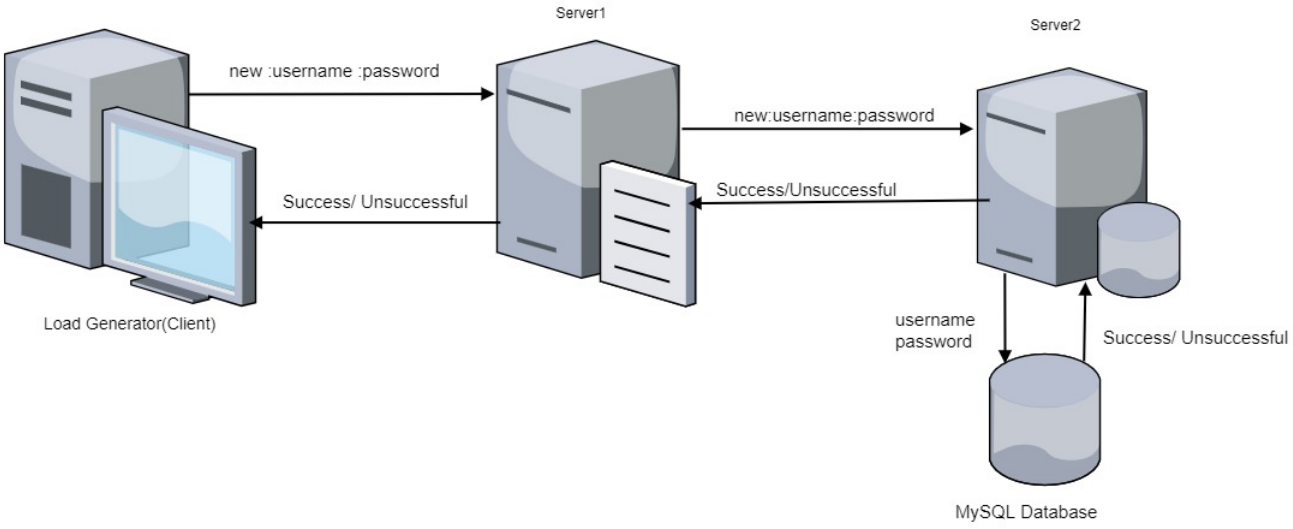

Figure 3: "New" request for adding user

- **To request file from server1**. In this case, "get" keyword followed by username and password will be sent to server1 (**get: uname: upassword**). Server1 will send "auth" keyword followed with username and password to Server2 . Server2 will check the database for user account.If user details are correct, server1 will be informed and server1 will request filename from user. Client will then send filename with extension to server1(File server) and then file will be sent to client if that file exists at server1.

  During load test, we generate request for fetching a file with username and password. Server1 forwards username and password to Server2

which authenticate the user . In this case,**there is file transfer (ie disk access at Server1) and new socket connection** is created between Client, Server 1 and server2 for each request. **Server2 will connect to mySQL database** for each request and authenticate the user.

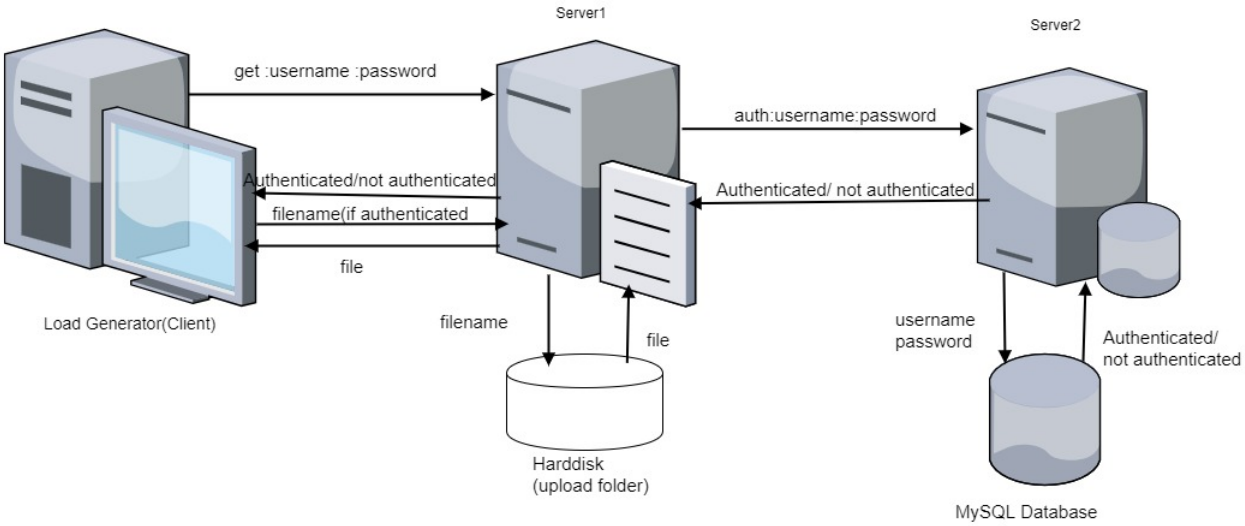

Figure 4: "Get" request for fetching file

3 Testing environment

Server1 (File server) and Server2 (Authentication server) run on same Virtual Machine. Client is running on host machine. Client connects to server machine over bridged network between host and VM. Server contains upload folder which contains files which client can download.

- **Client and Server Hardware Configuration**. The hardware specifications of server and client machine is as follows:

	Load Generator (Host)	**Servers (inside VM)**
Processor	4x Intel(R) Core(TM) i5-4440 CPU @ 3.10GHz	4x Intel(R) Core(TM) i5-4440 CPU @ 3.10GHz
Memory	7825MB	2048MB
Operating System	Ubuntu 16.04.3 LTS	Ubuntu 16.04.3 LTS
Architecture	x86_64	x86_64
CPU op-mode(s)	32-bit, 64-bit	32-bit, 64-bit
CPU(s)	4	2
On-line CPU(s) list	0-3	0-1
Thread(s) per core	1	1
Core(s) per socket	4	2
Socket(s)	1	1
Virtualization	VT-x	VT-x
L1d cache	32K	32K
L1i cache	32K	32K
L2 cache	256K	256K
L3 cache	6144K	6144K
NUMA node0 CPU(s)	0-3	0-1

- **Socket Tuning**. In order to make our server handle many concurrent connections, there is requirement to tune socket parameters. In listen system call in server, we define number of maximum incoming connections that can be handled. By default, in ubuntu this limit is 128 which is very less if we want to saturate the server.

  To view maximum connections handled by socket:
  cat /proc/sys/net/core/somaxconn
  128(default)

  listen(2) manual says - net.core.somaxconn acts only as upper boundary for an application which is free to choose any value less than this number.With a large number of clients communicating with server, there is requirement to increase this number. Eg. Command to increase the number to 30000 :

  sysctl -w net.core.somaxconn=30000
  Add net.core.somaxconn=30000 to /etc/sysctl.conf for it to become

permanent (be reapplied after booting)

Similarly, some other paarmeters that can be tuned are:

```
sudo sysctl net.ipv4.ip_local_port_range="15000 61000"
sudo sysctl net.ipv4.tcp_fin_timeout=10
sudo sysctl net.ipv4.tcp_tw_recycle=1
sudo sysctl net.ipv4.tcp_tw_reuse=1
sudo sysctl net.core.netdev_max_backlog=2000
sudo sysctl net.ipv4.tcp_max_syn_backlog=2048
```

- **Increasing ulimit parameters**. By default the number of open files for user in Ubuntu is 1024. We have to increase it to measure maximum throughput.This is done with the ulimit command:

  ```
  ulimit -a // see all the kernel parameters
  ulimit -n // see the number of open files
  ulimit -n 9000 // set the number open files to 9000
  ```

  To make these changes permanent, add the following line to /etc/security/limits.conf:
  useracct hard nofile 64000

- **Generating test files**. We have to create files of varying sizes to check system load for different file sizes.Since, we don't care about contents of file and only file size matters, we used /dev/urandom option to create file. The shell code for same is as follows:

  ```
  #! /bin/bash
  for n in 1..5000; do
  base64 /dev/urandom|head -c 200000000 > 200MB$n.txt
  done
  ```

4 Performance metrics

4.1 Test Load : "New" request to create account

As mentioned earlier, during this load test, we generate request for create new account.In this case, there is no file transfer (ie no disk access at Server1) . Server2 will connect to mySQL database to create an account.For each

N, we defined total runtime ie 100-120 seconds.Each thread sent request to create account till runtime is over. We counted the number of requests completed and based on this time, we calculated throughput as Total Number of requests/ Actual elapsed time. For each thread,we generated the random string for new user's username and password which was sent to server.

- Average Throughput and Average Response Time. The throughput increases till saturation point and then decreases and flattens out.

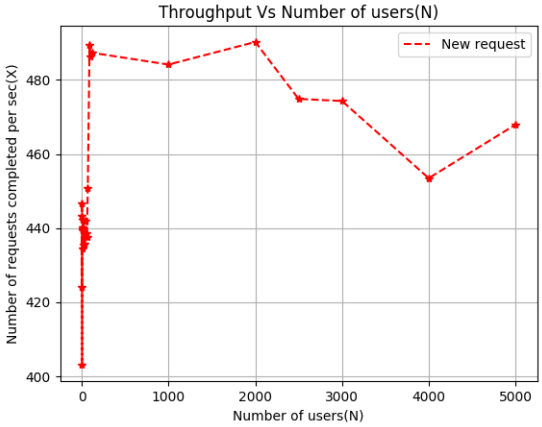

Figure 5: Throughput : New request

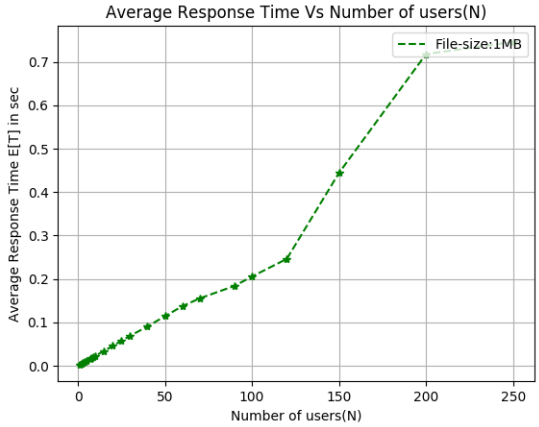

Figure 6: Average Response Time : New request

4.1.1 Identification of the bottleneck

Since there was no disk IO involved, we suspected bottleneck to either be network or CPU. Since servers were running on same VM with 2 cores and 2GB RAM, there was increase in CPU utilisation upto 40%. As it can be seen from figures below that at bottleneck, CPU utilisation has increased. We suspect that since for each new connection requesting new account, server2(backend server) forks new process in which MySQL connection is made to mySQL server,there is increase in utilisation of CPU resources as Server and mySQL process are among top processes in term of CPU utilization. Also, as we increased N, there were more SQL database connection errors.

Figure 7: CPU Utilisation at bottleneck using System monitor

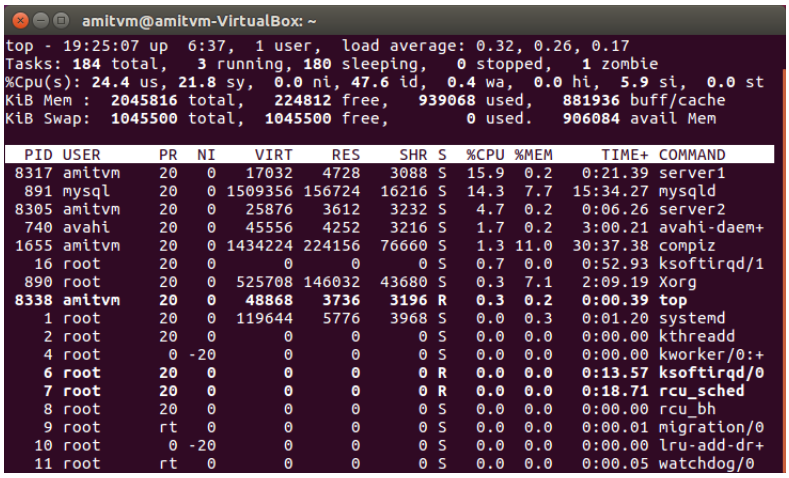

Figure 8: CPU Utilisation by server1, server2 and mysql processes at bottleneck

4.2 Test Load : Fetching same file per request

In this load test, each user thread fetched same file of given file size everytime.We generated one test files of given size(1KB, 1MB, 50MB, 500MB).

For each test, we defined total runtime ie 100-120 seconds.Each thread sent request to fetch file till runtime is over. We counted the number of requests completed and based on this time, we calculated throughput as Total Number of requests/ Actual elapsed time. We also made sure that file has already been accessed/requested once so that chances of file being already in disk buffer cache are high. We wanted to check how much throughput and response time we can achieve if file being already fetched.

4.2.1 File size: 1KB

When we used single file of 1Kb size (ie same load request) which load generator asks from file server, we got constant increase in throughput till we hit hardware bottleneck at load generator machine. Since file size is very small, it was already there in disk buffer cache and hence, there was no disk access. However, as we increased the number of users, the CPU utilisation kept increasing and we reached the point where throughput was not increasing further.

- Average Throughput and Average Response Time

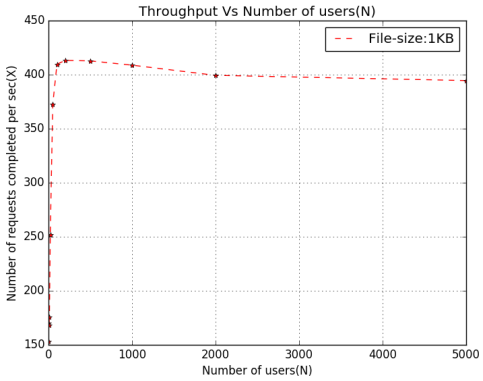

Figure 9: Throughput : File size 1KB

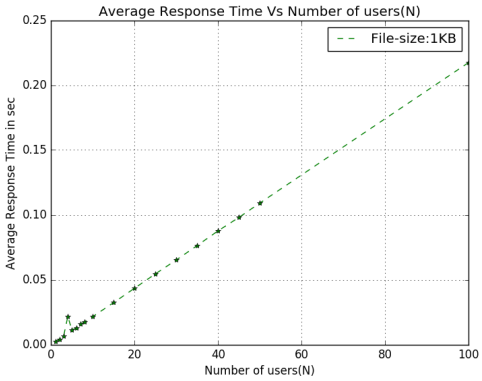

Figure 10: Average Response Time : File size 1KB

4.2.2 File size: 10KB

- Average Throughput and Average Response Time

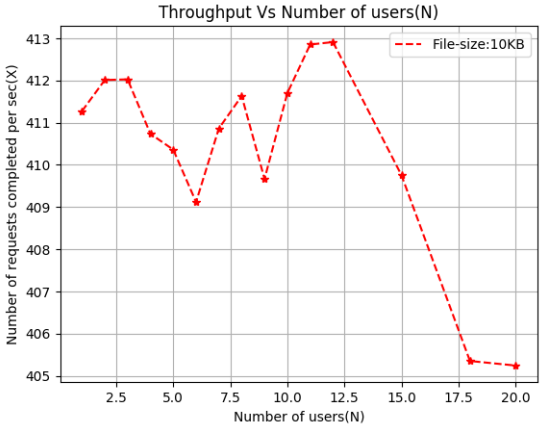

Figure 11: Throughput : File size 10KB

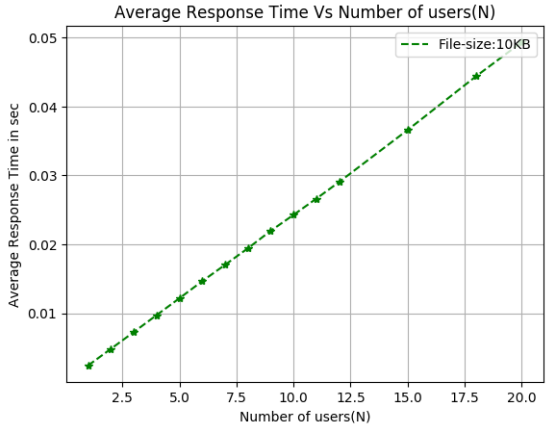

Figure 12: Average Response Time : File size 10KB

4.2.3 File size: 50KB

- Average Throughput and Average Response Time

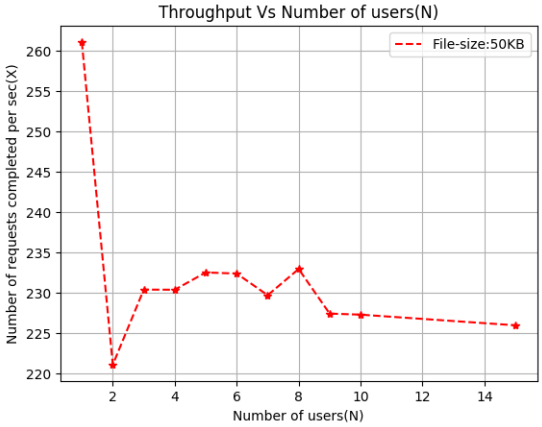

Figure 13: Throughput : File size 50KB

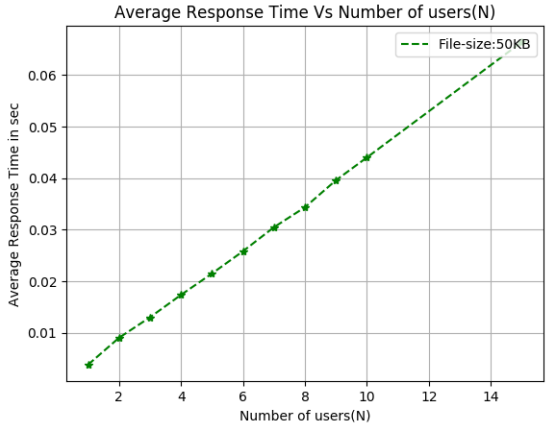

Figure 14: Average Response Time : File size 50KB

4.2.4 File size: 500KB

– Average Throughput and Average Response Time

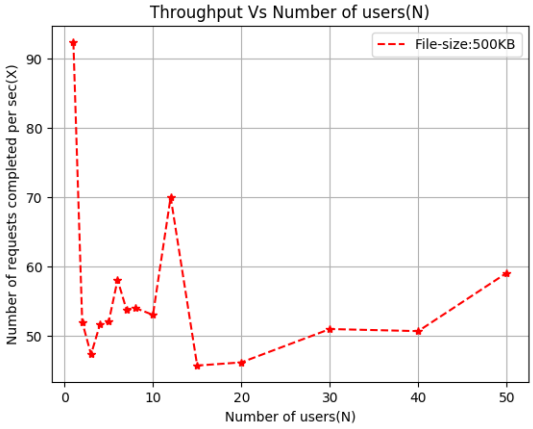

Figure 15: Throughput : File size 500kB

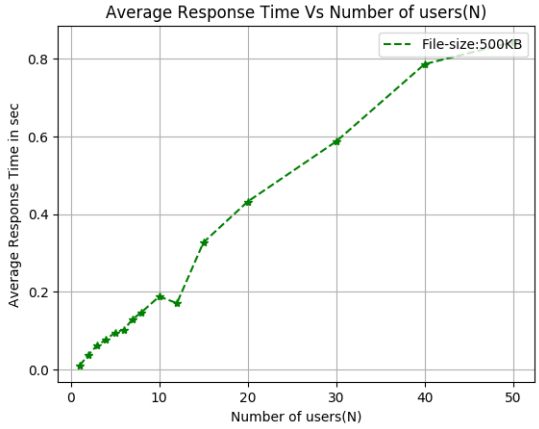

Figure 16: Average Response Time : File size 500kB

4.2.5 File size: 1MB

– Average Throughput and Average Response Time

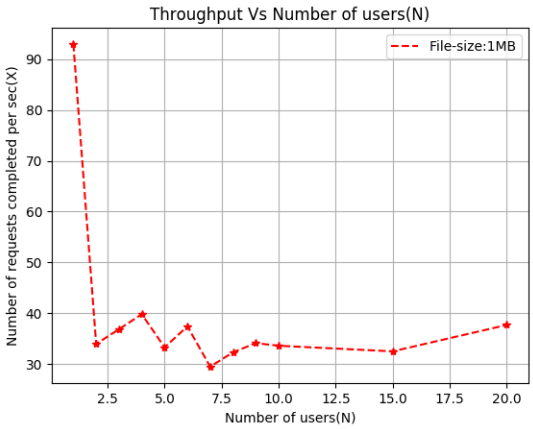

Figure 17: Throughput : File size 1MB

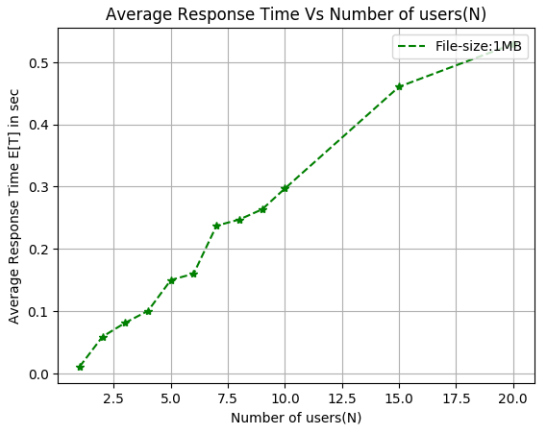

Figure 18: Average Response Time : File size 1MB

4.2.6 File size: 50MB

– Average Throughput and Average Response Time

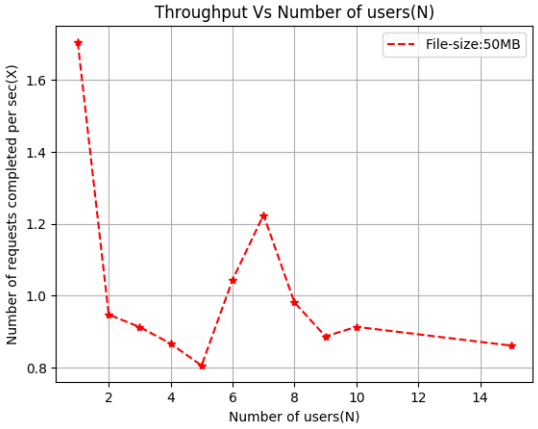

Figure 19: Throughput : File size 50MB

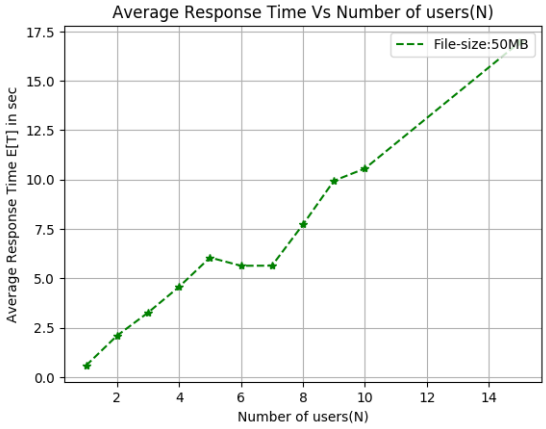

Figure 20: Average Response Time : File size 50MB

4.2.7 File size: 500MB

When we used single file of 500MB size , we saw even though we were fetching same file for multiple times for multiple users, there was increase in disk access.

– Average Throughput and Average Response Time

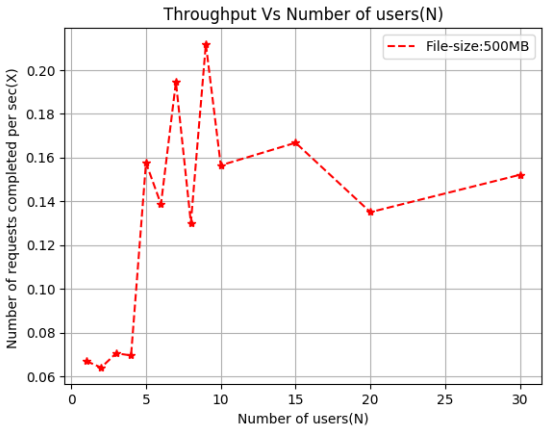

Figure 21: Throughput : File size 500MB

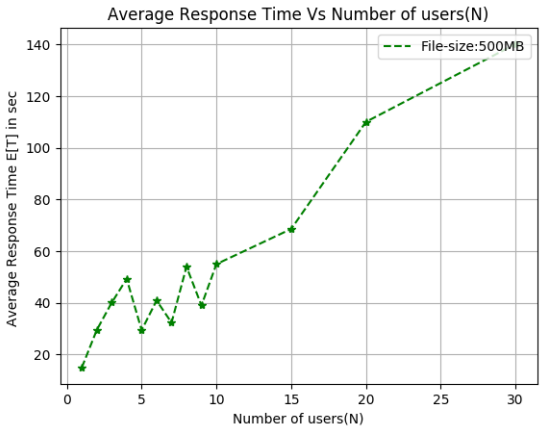

Figure 22: Average Response Time : File size 500MB

4.2.8 Identification of the bottleneck

When we used files with small file size, we got constant increase in throughput till we hit hardware bottleneck at load generator machine.The throughput and response time for such files was very high since file size was small, it was already there in disk buffer cache and hence, there was no disk access. However, as we increased the number of users, the CPU utilisation kept increasing and we reached the point where throughput was not increasing further. Once initial trials gave us the maximum throughput and N*, we again did trails at saturation point to identify the bottleneck resource. We used nmon and System monitor tools to identify the resource. Also top tool was used to identify the processes consuming maximum CPU resources. The following screen-shots for different file sizes and number of users(N) help us to identify the hardware bottleneck in this case as CPU.

CPU as bottleneck for file size 1KB at saturation point.

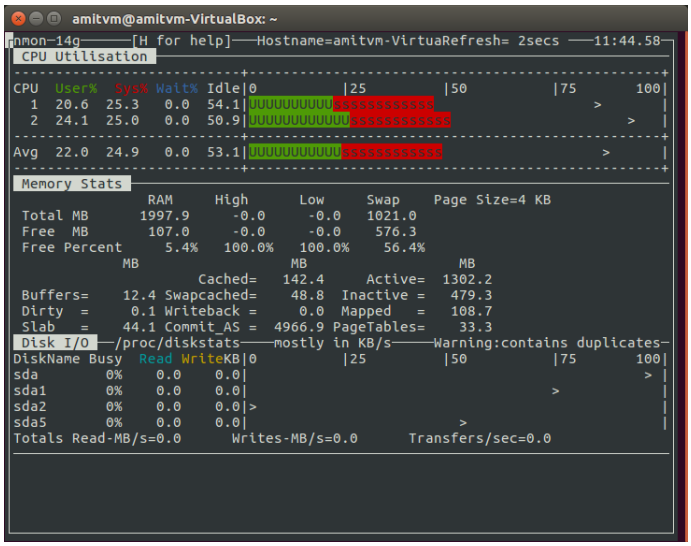

Figure 23: Disk and CPU Utilisation for file size 1KB at saturation

For file size=50KB, CPU utilisation before reaching saturation point for N=1.

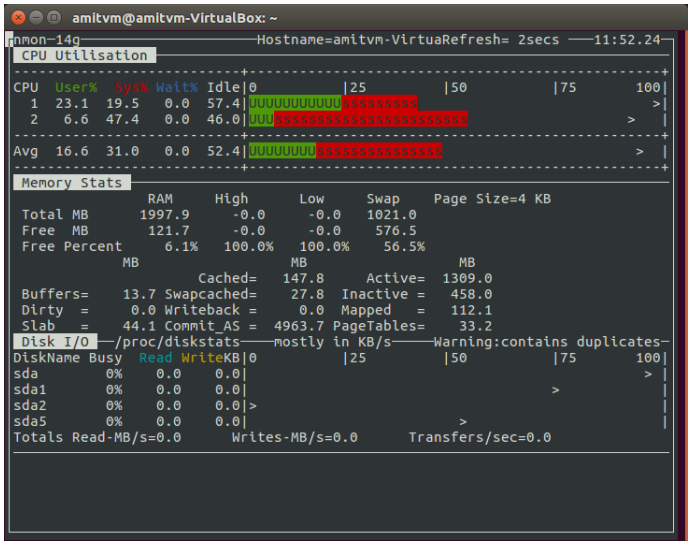

Figure 24: Disk and CPU Utilisation for file size 50KB before saturation point using nmon

CPU utilisation increasing sharply at saturation point.

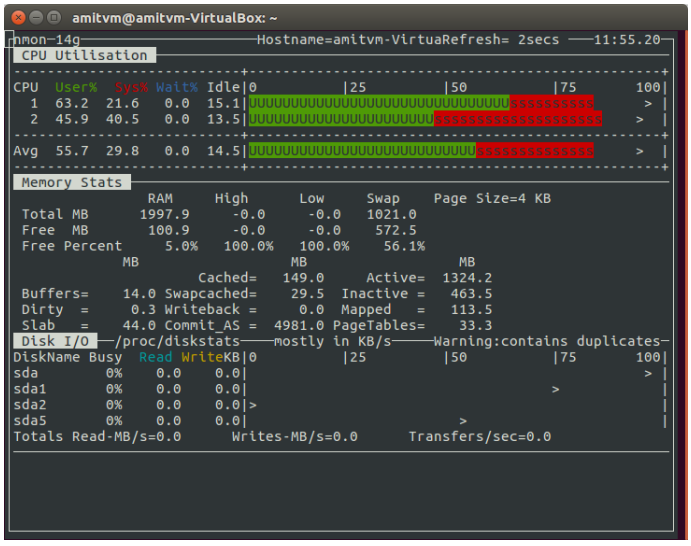

Figure 25: Disk and CPU Utilisation for file size 50KB after saturation using nmon

Figure 26: CPU Utilisation for file size 50KB at saturation using system monitor

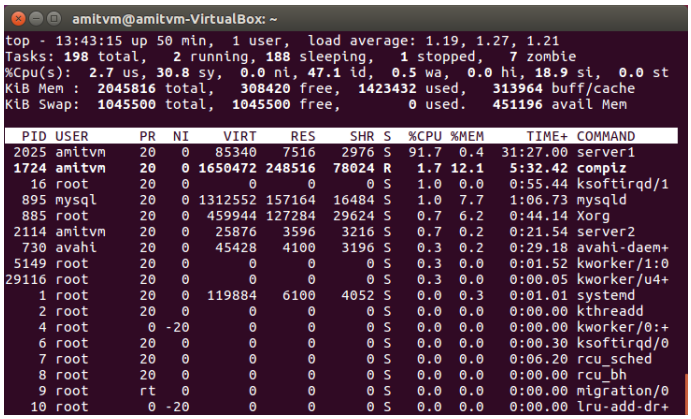

Figure 27: CPU utilisation for file size 1MB at saturation

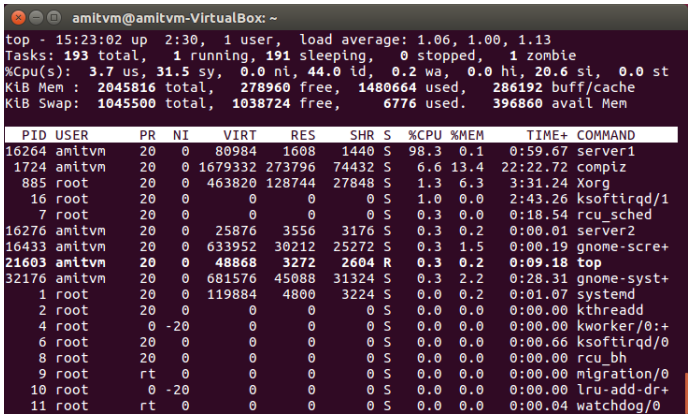

Figure 28: CPU utilisation for file size 50MB at saturation

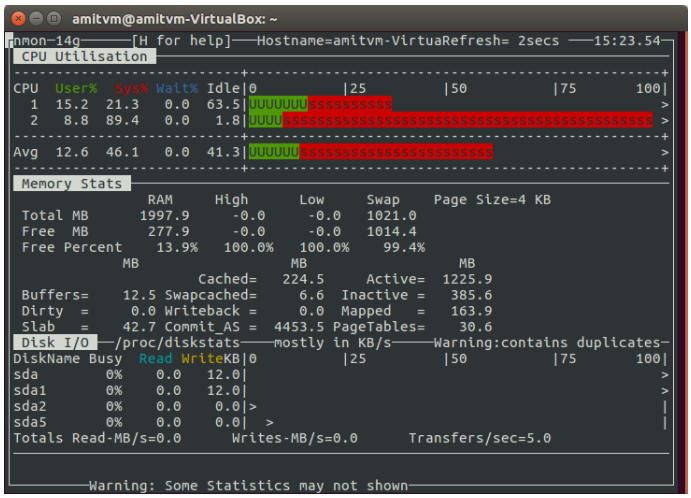

Figure 29: CPU and disk utilisation for file size 50MB at saturation

However, for large files, we saw even though we were fetching same file for multiple times for multiple users, there was increase in disk access as can be seen from figure below. The reason for same can be that such large files may not fit into disk buffer cache and hence has to be accessed from disk as different users may be reading different portions of the same file.

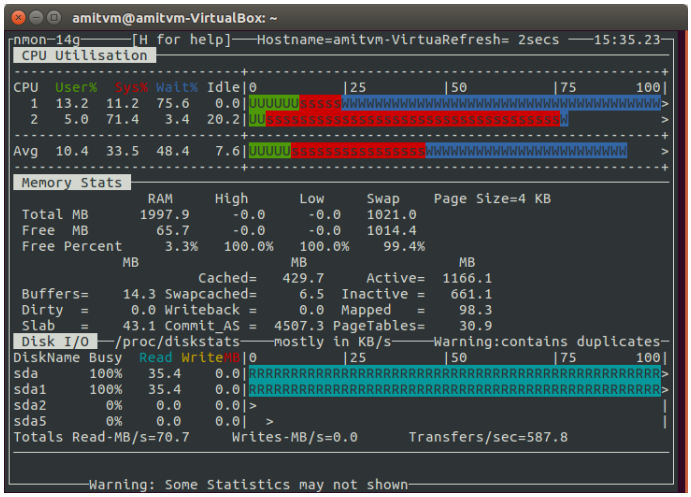

Figure 30: CPU and disk utilisation for file size 500MB using nmon

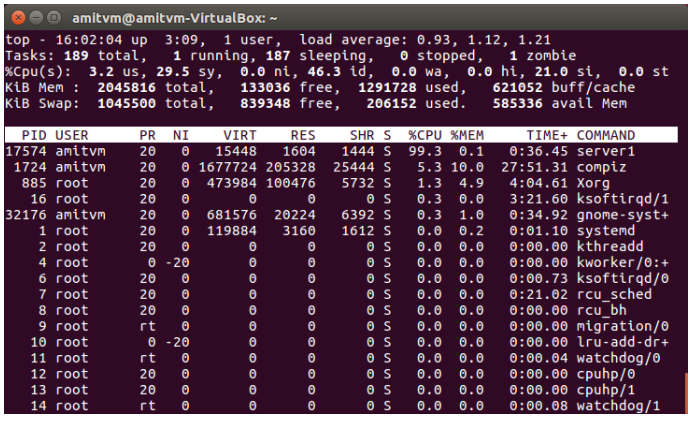

Figure 31: CPU for file size 500MB using top

However, it was later observed, after certain value of N, the disk access minimised and CPU utilisation increased. The reason might be that by that time file would have been fully cached into memory and disk access reduced.

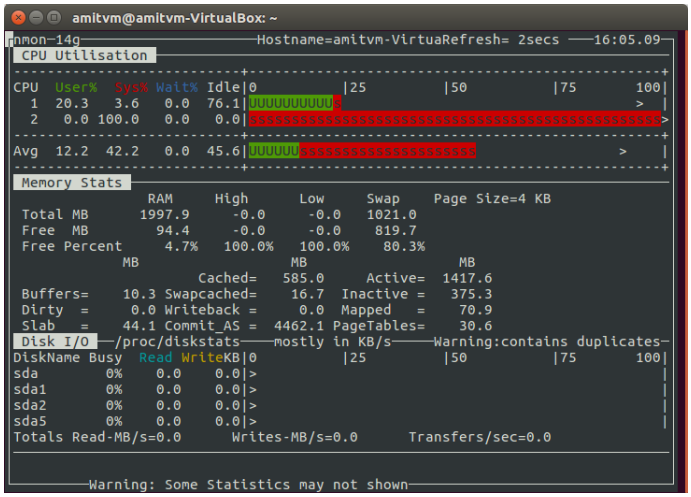

Figure 32: CPU and disk utilisation for file size 500MB using nmon

4.3 Load : Fetching different files per request

In this load test, we each user thread fetched different file of given file size everytime.We generated large number of test files of given size(1KB, 500KB, 10MB, 50MB, 500MB). There two approaches to fetch different files from server by different user threads so that probability of file being in cache is reduced.

In one option,for each user we defined number of total fetch file requests.For instance, 2000 file request for each user. So, N=1,there were total 2000 requests,for N=2,total 4000 requests, for N=10, total 20000 requests and so on. The total time taken to complete these requests was calculated.And based on this time, we calculated throughput as Total Number of requests/ Total time taken.This approach helped us to ensure that total bytes of data downloaded by each user remained same for testing. In order to ensure that no user thread fetches the already fetched file and read the file from disk, we generated file name for each request as follows:

```
int r = (long(threadid)* 2000) +num_req+1;
strcpy(file_name,"500KB");
sprintf(count, r);
strcat(file_name,count);
strcat(file_name,".txt");
num_req++;
```

This ensured that there is offset of 2000 between files demanded by each thread as per thread id and the same file is not requested again by some other thread. But, we had to generate 2000*N files for such test case.

Other approach was to randomly generated the file name seeded by unique thread ID and access these random files by specifying run time for the test:

```
srand(long(threadid));
int r = rand()mod2000 +1;
strcpy(file_name,"500KB");
sprintf(count, r);
strcat(file_name,count);
strcat(file_name,".txt");
timer++;
```

We did trials for both approaches. For files with small size we used approach I as there were more chances of same files being fetched by threads to be cache as file size was less and possible to generate large number of small szied files. For larger sized files,the results were almost same. Also, after every one test for some value of N for given file, we cleared the buffer cache using following commands:-

sync; sudo sh -c "echo 3 > procsysvmdrop_caches"

Note:(Sharp fall in throughput/response time figures ahead is because of scale used)

4.3.1 File size: 1KB

- Average Throughput and Average Response Time. The throughput increases till saturation point and then decreases a little and

flattens out.

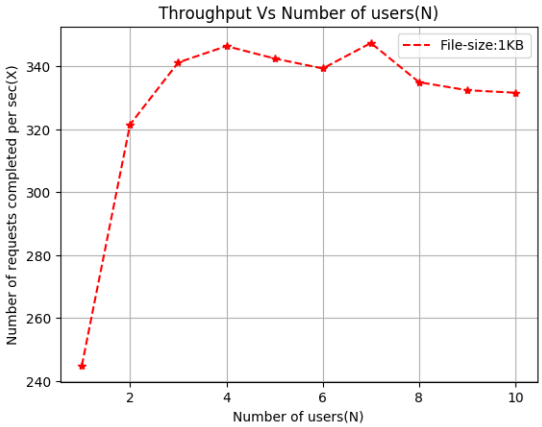

Figure 33: Throughput : File size 1KB

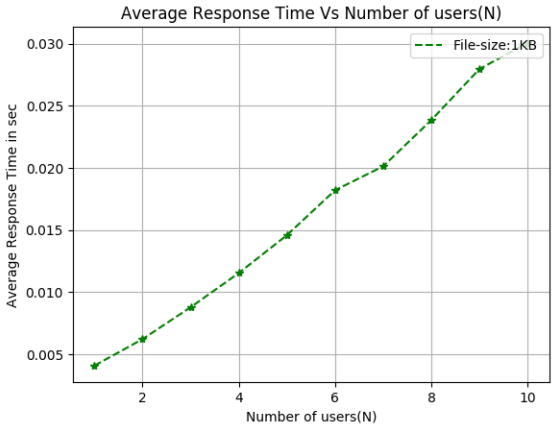

Figure 34: Average Response Time : File size 1KB

4.3.2 File size: 100KB

– Average Throughput and Average Response Time. The throughput shows a dip as increase N from 1 to 2 but after that it flattens but then decreases.

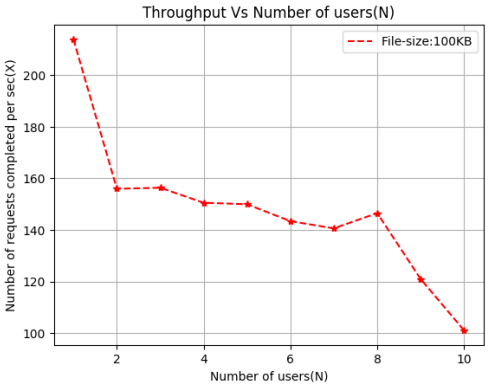

Figure 35: Throughput : File size 100KB

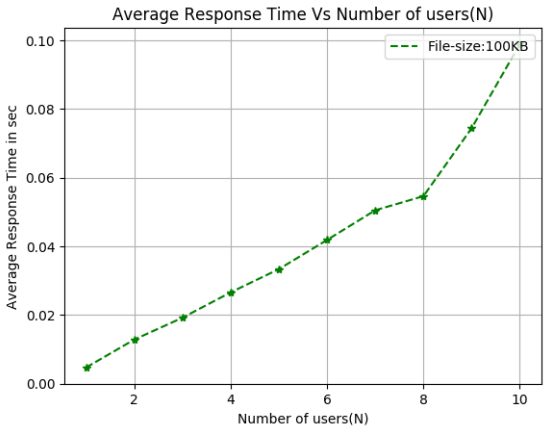

Figure 36: Average Response Time : File size 100KB

4.3.3 File size: 1MB

– Average Throughput and Average Response Time. Here, maximum throughput was achieved for N=1. As we kept increasing N, throughput decreased and then flattened out.

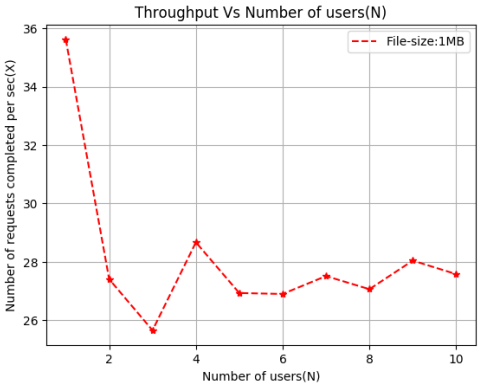

Figure 37: Throughput : File size 1MB

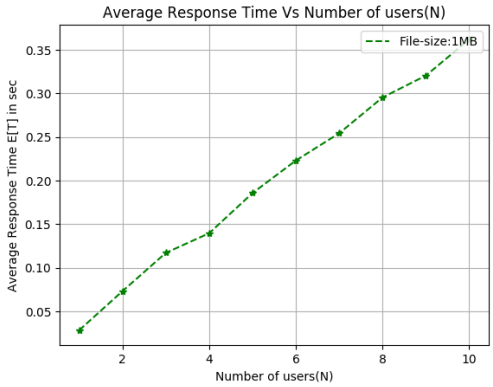

Figure 38: Average Response Time : File size 1MB

4.3.4 File size: 10MB

– Average Throughput and Average Response Time. Here, maximum throughput was achieved for N=1. As we kept increasing N, throughput decreased and then flattened out.

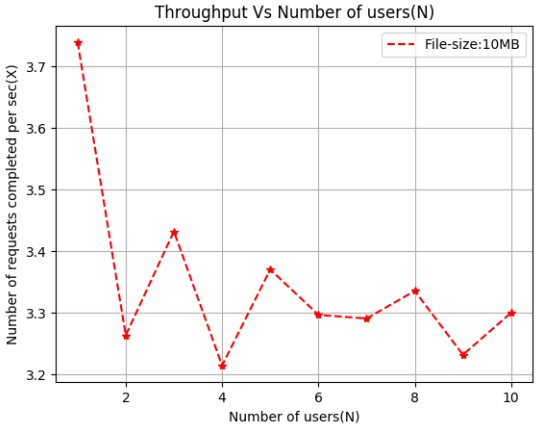

Figure 39: Throughput : File size 10MB

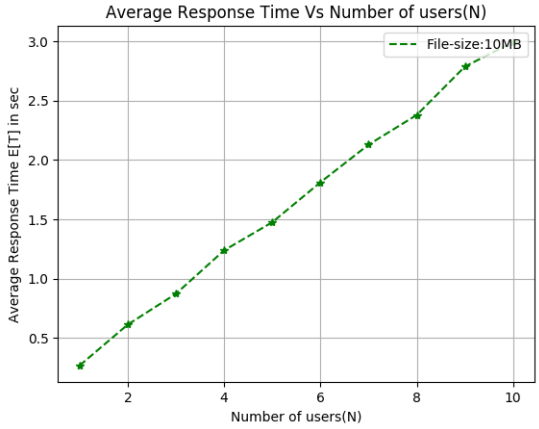

Figure 40: Average Response Time : File size 10MB

4.3.5 File size: 50MB

– Average Throughput and Average Response Time. Here also, maximum throughput was achieved for N=1. As we kept increasing N, throughput decreased and then flattened out.

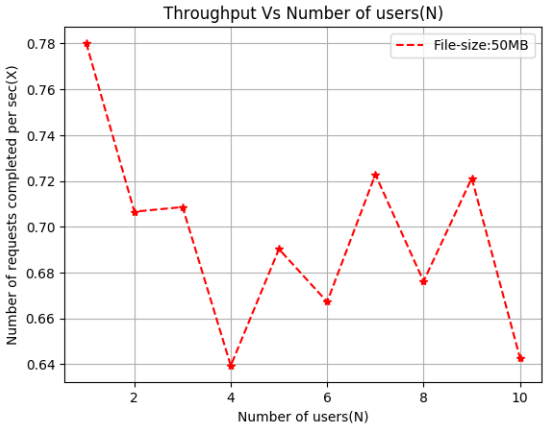

Figure 41: Throughput : File size 50MB

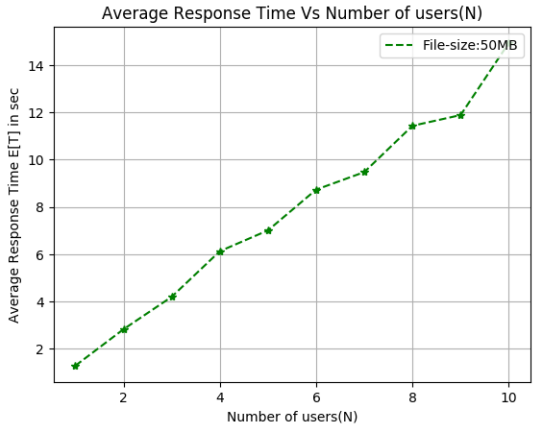

Figure 42: Average Response Time : File size 50MB

4.3.6 File size: 500MB

– Average Throughput and Average Response Time. The throughput increases till saturation point, here N=2 and then it decreases and flattens out. (Sharp fall in figure is because of scale used)

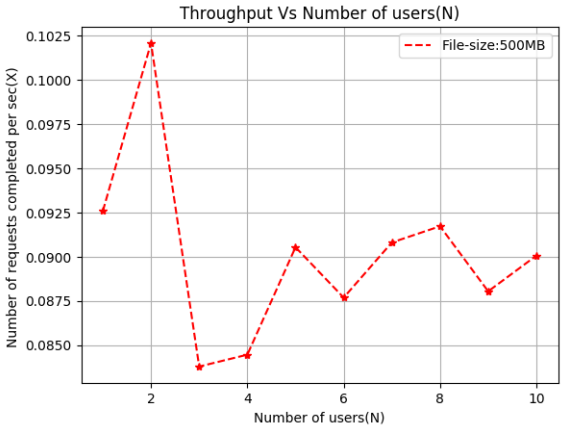

Figure 43: Throughput : File size 500MB

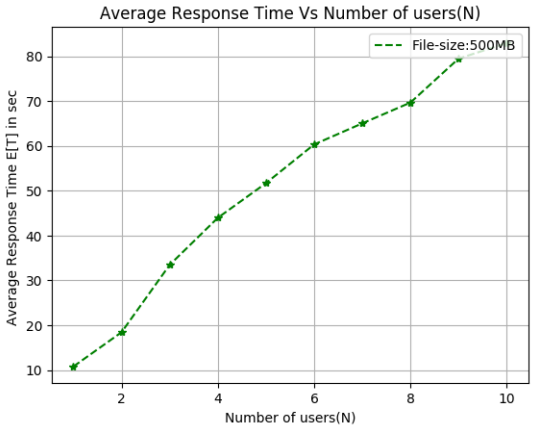

Figure 44: Average Response Time : File size 500MB

4.3.7 Identification of the bottleneck

Once initial trials gave us the maximum throughput and N*, we again did trails at saturation point to identify the bottleneck resource. We used nmon , iostat and System monitor tools to identify the resource.

```
Linux 4.10.0-28-generic (amitvm-VirtualBox)     Saturday 07 October 2017     _x86_64_    (2 CPU)

Device:         rrqm/s   wrqm/s     r/s     w/s    rkB/s    wkB/s avgrq-sz avgqu-sz   await r_await w_await  svctm  %util
sda               3.98    26.69   15.21    2.81   688.80   124.66    90.29     0.28   15.59    8.45   54.24   2.32   4.18

Device:         rrqm/s   wrqm/s     r/s     w/s    rkB/s    wkB/s avgrq-sz avgqu-sz   await r_await w_await  svctm  %util
sda               0.00    1.40   35.20    0.60   141.60     8.00     8.36     0.34    9.36    9.18   20.00   1.27   4.56

Device:         rrqm/s   wrqm/s     r/s     w/s    rkB/s    wkB/s avgrq-sz avgqu-sz   await r_await w_await  svctm  %util
sda               0.00    0.60   22.80    0.40    91.20     4.00     8.21     0.15    6.31    6.25   10.00   0.93   2.16

Device:         rrqm/s   wrqm/s     r/s     w/s    rkB/s    wkB/s avgrq-sz avgqu-sz   await r_await w_await  svctm  %util
sda               0.00    0.00  180.20    0.00  1045.60     0.00    11.60     0.69    3.81    3.81    0.00   2.04  36.80

Device:         rrqm/s   wrqm/s     r/s     w/s    rkB/s    wkB/s avgrq-sz avgqu-sz   await r_await w_await  svctm  %util
sda               0.00    1.20  181.40    0.40   725.60     6.40     8.05     0.30    1.65    1.64    8.00   1.39  25.28

Device:         rrqm/s   wrqm/s     r/s     w/s    rkB/s    wkB/s avgrq-sz avgqu-sz   await r_await w_await  svctm  %util
sda               0.00    1.80   10.00    0.40    40.00     8.80     9.38     0.02    2.15    1.44   20.00   1.46   1.52
```

Figure 45: Disk Utilisation using iostat for file size 1KB at saturation

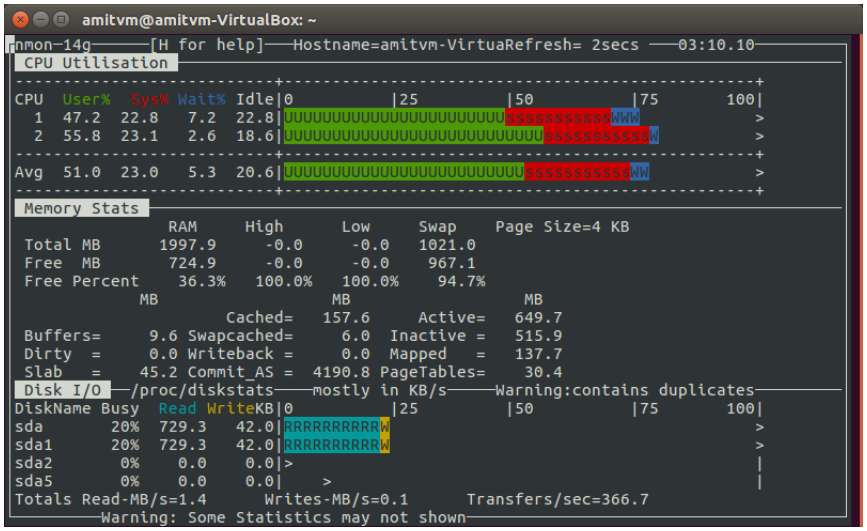

Figure 46: Disk and CPU Utilisation using nmon for file size 1KB at saturation

As file size increases and we increase number of users N, CPU starts spending more time in waiting state for disk IO operations.

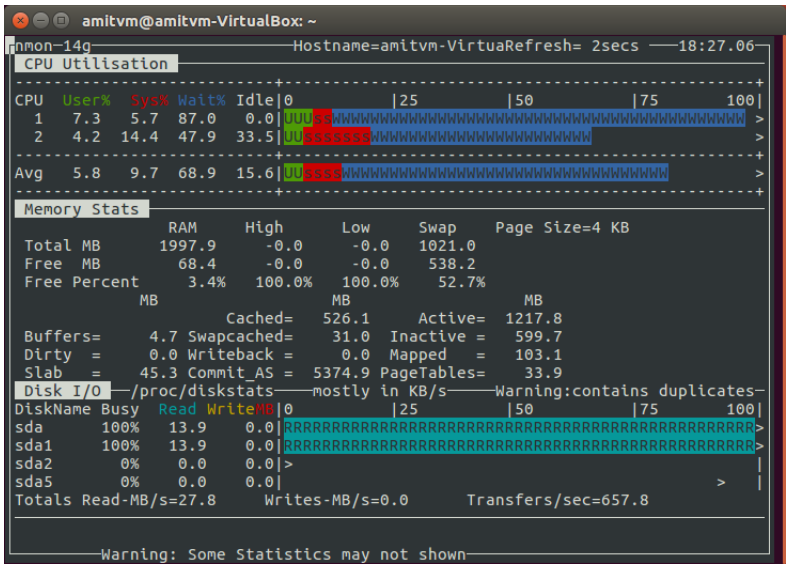

Figure 47: Disk Utilisation using nmon for file size 1MB after saturation

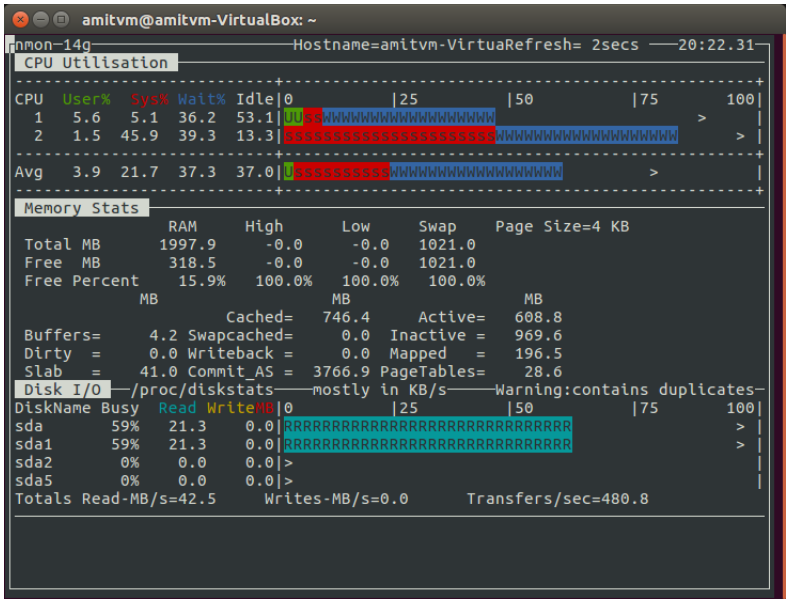

Figure 48: Disk and CPU Utilisation for file size 10MB at saturation

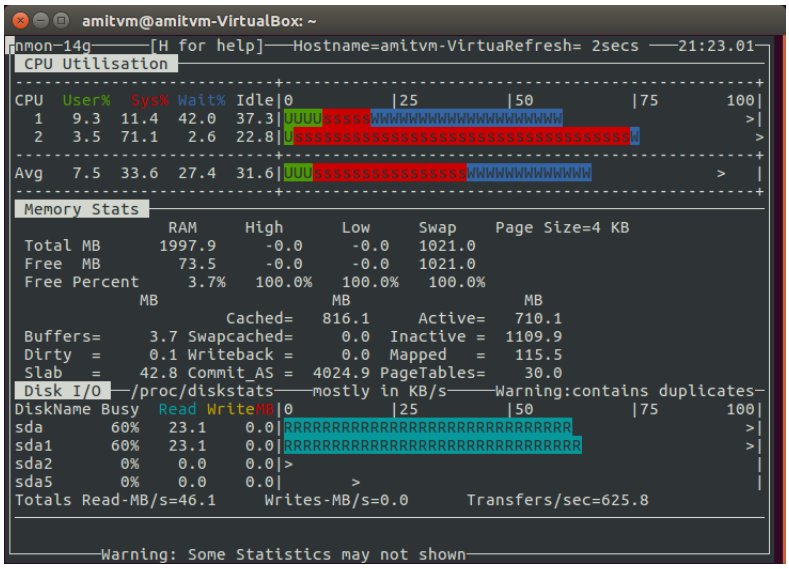

Figure 49: Disk and CPU Utilisation for file size 50MB at saturation

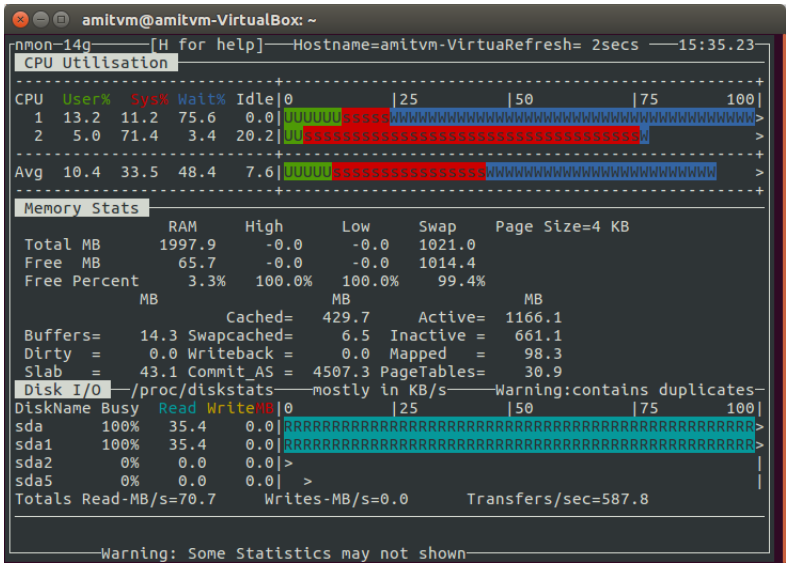

Figure 50: Disk and CPU Utilisation for file size 500MB after saturation

4.4 Profiling the code

In order to find which portion of code/function, our server1 is spending most of the time, we used valgrind tool.

valgrind –tool=callgrind ./server1 10000

It generated a file called callgrind.out.12445. We used kcachegrind tool to read this file and get graphical analysis.

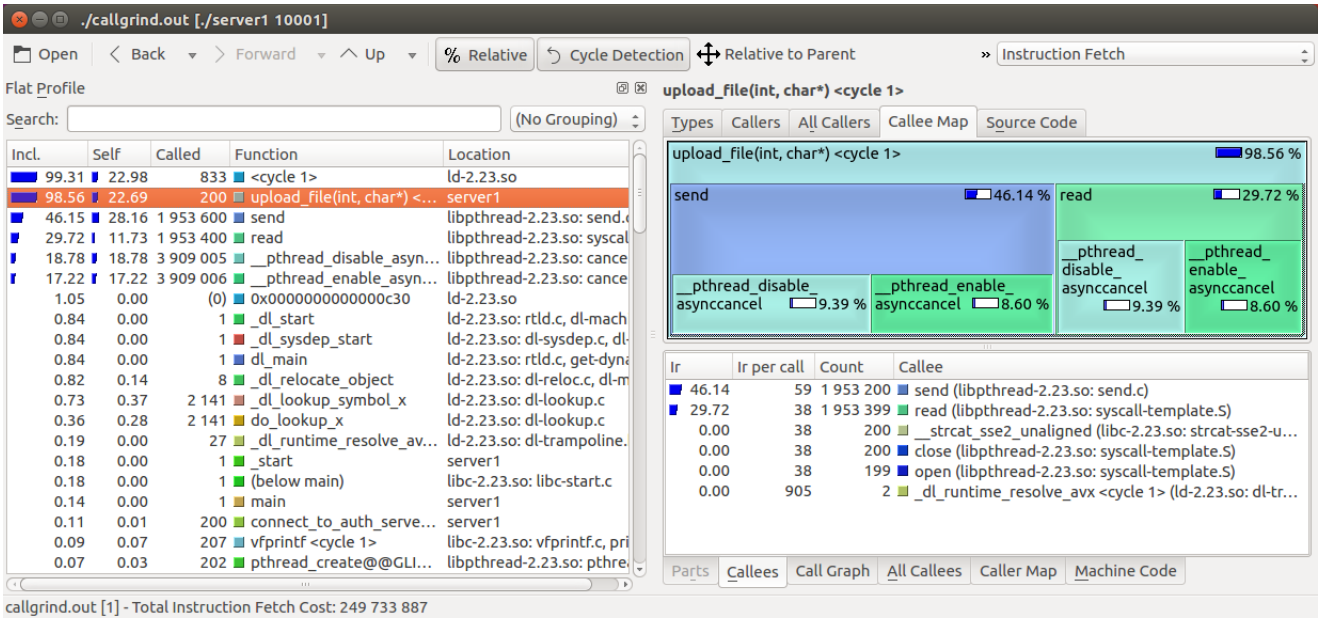

Figure 51: Code profiling

We can see in the profile output that our file server spends maximum time in "**upload_file**" function, ie reading the file and sending the file to client.

5 Summary of results

We saw while fetching a new file, throughput increases till saturation point and then decreases a little and then flattens out for most cases. For most cases, maximum throughput was achieved at N=1 or smaller values of N while fetching new file. The reading new file from disk acts as bottleneck for server1. As we kept increasing N, the throughput decreased and flattened out.As file size keeps increasing maximum throughput(Average number of requests completed per second) decreases and service time increases due to more disk IO. Also, our server is 100% saturated because there is no wait time so as one request completes new request is sent,keeping server busy. The relative comparison between throughput and response time for fetching different files from disk as we increase N and file size:

– Average Throughput

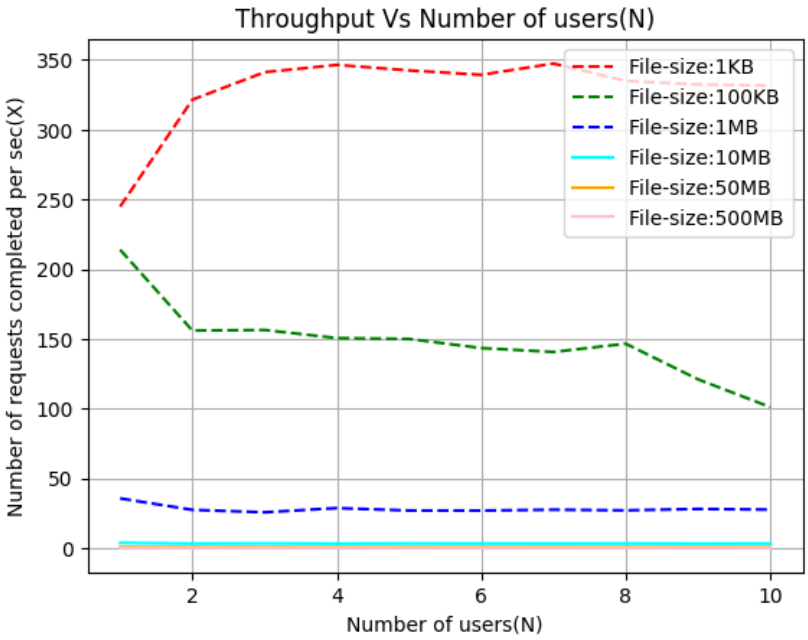

Figure 52: Comparison in throughput with different file sizes

– Average Response Time

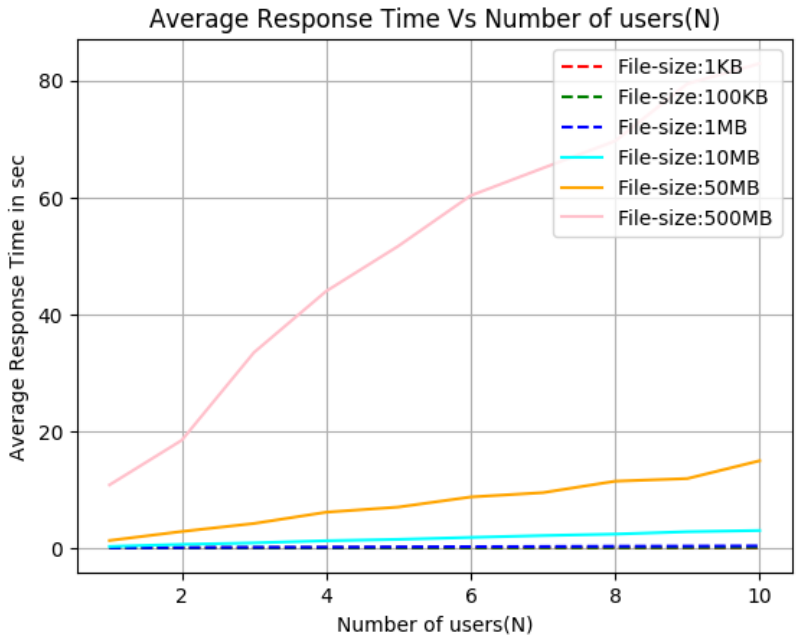

Figure 53: Comparison in Average Response Time with different file sizes

Load type	File size	N*	Mean Response Time at N=N*	Max Throughput at N=N*	Bottleneck
new		120	0.245996	487.341	CPU
get (diff file)	1KB	4	.011538	346.459	Disk I/O
get	100KB	1	0.00467084	213.901	Disk I/O
get	1MB	1	0.0280648	35.6256	Disk I/O
get	10MB	1	0.26741	3.73949	Disk I/O
get	50MB	1	1.28167	0.780229	Disk I/O
get	500MB	2	18.4423	0.102084	Disk I/O

6 Implementation details

6.1 Server2 : Authentication server

Server2 (Backend authentication server) receives username and password from Server1 for either creating new account or to authenticate

the user.It will have MySQL server installed. Step by step procedure for installation is as follows: Two shell scripts provided in code folder:

installmysql.sh

importdatabase.sh
Makefile has also been provided to compile C++ code. Steps to run Server2:

./server2 Listening-port

6.2 Server1 : File server

Server1 receives username and password from client for either creating new account or to authenticate the user for fetching file and will send this username and password to Server2(Authentication server). Once authenticated by Server2, Server1 will get filename from client and will provide file to client if file exists. Makefile has been provided to compile C++ code. Steps to run Server1:

./server1 Listening-port

6.3 Client : Load Generator

Client sends user name and password to server for either creating new account or to authenticate himself for fetching file. Once authenticated, client will send filename to server1 and will download file from server1 if file exists. Makefile has been provided to compile C++ code. Steps to run client/load generator:

./client Server1_IP Server1_Listening-port request_type

7 Submission folder

There are three folders inside : client, server1 and server2. The folder/directory structure and files are listed below:

7.1 Client

- Makefile
- client.cpp : Load generator (with Number of request as input)
- client_timer.cpp : Load generator (with run time as input)
- "downloads" folder : to store received files.

7.2 Server1

- Makefile
- server1.cpp : File server code.
- "uploads" folder : to store files available for sharing/download.
- filecreate.sh : To create files with random content of specified size.
- clearbuffer.sh : To clear contents of cache buffer.
- tunesocket.sh : To tune the socket parameters.

7.3 Server2

- Makefile
- server2.cpp : Authentication server code.
- installmysql.sh : script to install MySQL server and required libraries for header files.
- importdatabase.sh : script to import database and user table from cs744.sql file.
- cs744.sql : contains MySQL database and table for import.