File Backup System
CS744 Project Report
Department of CSE, IIT Bombay

Kumar, Vikas
163059007@iitb.ac.in

Sen, Satyaki
16305R009@iitb.ac.in

5 November 2017

Contents

1 Introduction

The project File Backup System is used to backup the client files on the remote server which we will call as backend server via frontend server. The server(frontend server or authentication server) offers various options to client using which the client makes a request and gets back response from the server. There are 10 types of requests supported by the server as follows:

1. Login
2. Sign Up
3. Check File System
4. Upload Files
5. Download File
6. Share files
7. Download a shared file
8. Check shared files
9. Delete file
10. Logout

2 Architecture of System

The system follows a 3 tier (Client - Front-end Server - Backend Server) architecture model. All the requests are handled by the front-end server and it is the only point of contact for the client. The front-end server performs authentication check with a file database for every client request and fetches or stores files on the backend server. Once the Authentication check is successful the front-end server assigns a random session ID to the client. Now for each subsequent request from the client the front-end server authenticates client with session ID and then only serves the request.

1. **Phase 2 design : Multithreaded Server**

   To allow multiple client connections, the server program is converted to a multithreaded server program. Now for each request the front-end server spawns a new thread using pthread library. If Upload request is received then first the file is stored on front-end server temporarily and then a pthread is created to transfer it to backend and once the transfer completes the file is removed from the front-end server and a reply to client is sent. Similarly, in case of Download request,if the session check succeeds,

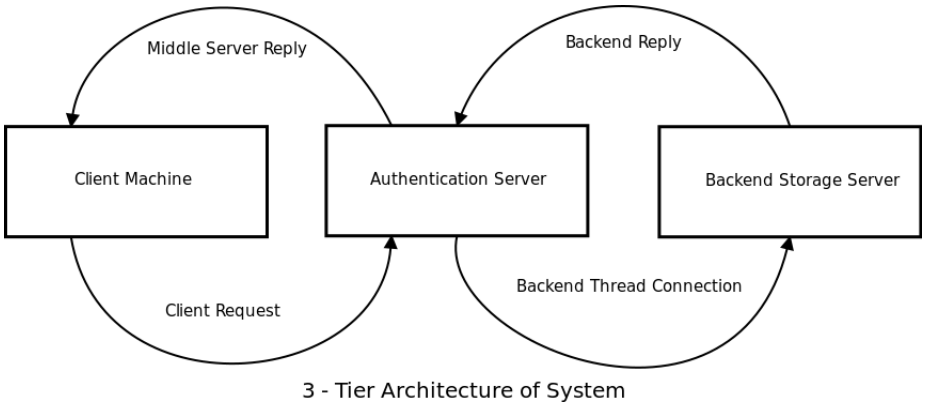

Figure 1: Architecture of System.

the file is first fetched from backend server to the front-end server,stored temporarily and then response is sent back to client.Finally the file is removed from the front-end server once it is transferred to the client.

2. **Phase 3 design : Non-blocking event driven server**

   To allow multiple client connections,the front end server program has used event driven non-blocking I/O.We have used Epoll to achieve event driven I/O mechanism.All other implementation of client and backend server is same a phase 2 design.

3 Design of performance testing

We perform the performance testing of the front-end server by simulating the single client to send multiple request in parallel to the front-end server using the Load generator as described in section 4. To simulate the client program to work as load generator we use pthread library. We perform an closed loop testing in which we spawn a fixed N number of parallel threads and run each thread for a fixed time(In case of phase 2) or for fixed number of requests(In case of phase 3) in our experiment.

3.1 Experimental Setup

We use 1 Ubuntu 16.04 Virtual Machine with 3 cores and 6 GB RAM to run our backend server. To run our front-server we have used 1 physical machine with 1 core and 8 GB RAM. The third physical machine is Ubuntu 16.04 machine with 16 GB physical memory and 8 cores CPU which we have used to run the Load generator.

4 Load Generator

We have used pthread library to make the client program behave as load generator by spawning N (which is given as argument) parallel threads for execution. Each thread runs for 5 minutes and makes multiple Download requests of 10KB and 5MB files to server. In order to ensure that the load generator is not bottleneck we measure the performance of the system by using 'htop' and 'iotop' programs to measure the cpu utilization and I/O utilization

We have also made sure that our load generator never becomes the bottleneck.For that reason we have run our Load generator on a machine with Total 8 cores and 16 GBs of physical memory.

In order to calculate total number of requests made by all the Users(Threads) we keep a local counter for every thread and one global counter.We count the number of requests made by one single thread in the local counter for that thread and add it to the global counter.

In order to calculate the response time of a request we have take the total time from the time when the client has generated a request to the time the client gets back a response from the front-end server.

To calculate the average throughput we have divided the total number of requests by the total execution time of all threads.

To calculate the average response time we got the total response time for all requests of all the threads and divided it by the total number of requests.

4.1 Scripts Used

For the load testing we have created a shell script which will call the load generator with different parameters. Here the parameters are:

1. Number of threads(or number of users)
2. Time of execution for each thread

The shell script will gradually increase the number of threads with a given execution time for each thread and will run the load generator.

In order to ensure that the file is written directly to disk and not cached in the disk buffer cache at the front-end server we run a cron job to clear disk buffer cache every 1 minute and sync the memory contents with disk.

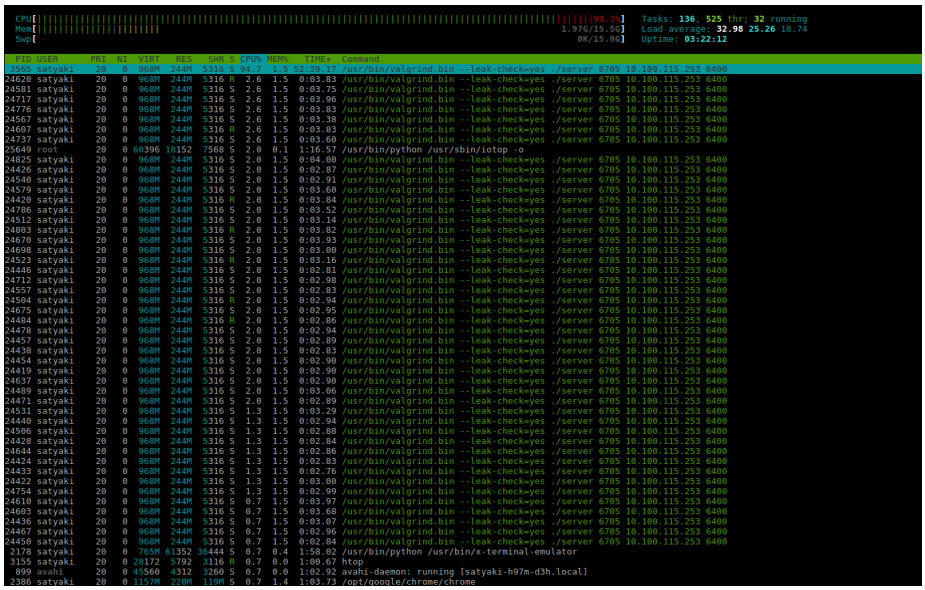

Figure 2: CPU Bottleneck.

5 Phase 2 performance analysis:

5.1 Two Different Scenarios for Generating Load

We have tested the load generator for two different scenarios.

Scenario 1 : We have populated multiple files of size 10KB at backend server.The Load Generator will run multiple threads for downloading random files of size 10 KB. Here we did not clear the disk buffer cache. So that the process mainly becomes an CPU bound process. As per the results it shows that in this case the CPU is becoming the bottleneck as shown in Fig.2. But as the disk buffer is caching the small files the I/O is not the bottleneck.

Scenario 2 : For case 2 we have populated multiple files of size 5MB. We are running a script at front-end server to continuously write the memory contents directly to Disk and clear the disk cache instantaneously. Here as the file size is much larger and we are clearing the buffer cache as well the I/O usage is also becoming high. And after a certain number of threads the usage of I/O becomes more than 90% as it can be seen in Figure 3.

5.2 Graphs

We plot two types of graph for the upload file request as given below with increasing value of load.

1. Average Throughput of System for scenario 1:

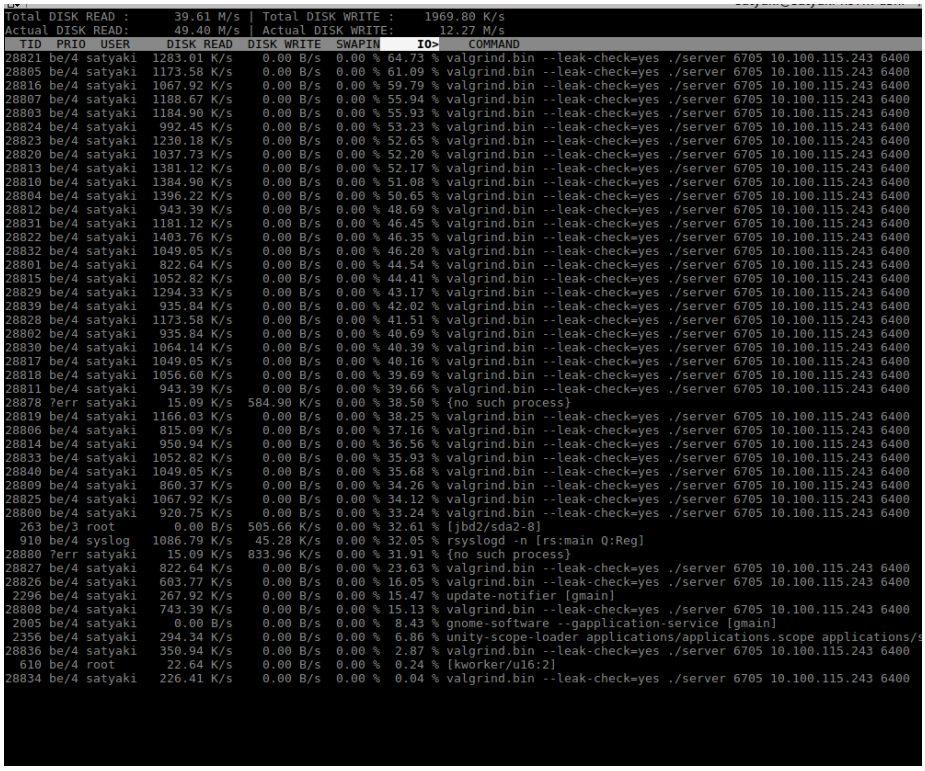

Figure 3: Disk I/O Bottleneck for 5MB File Download.

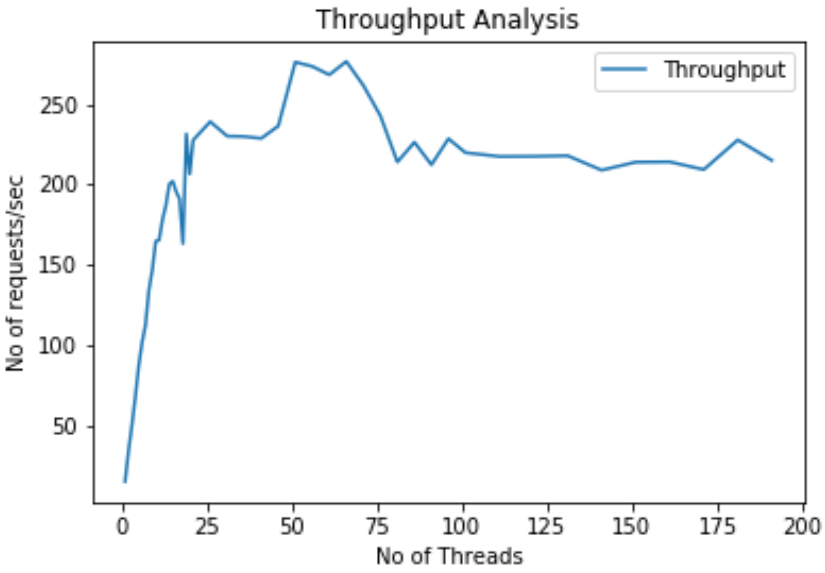

Figure 4: Throughput Graph for 10KB File Download.

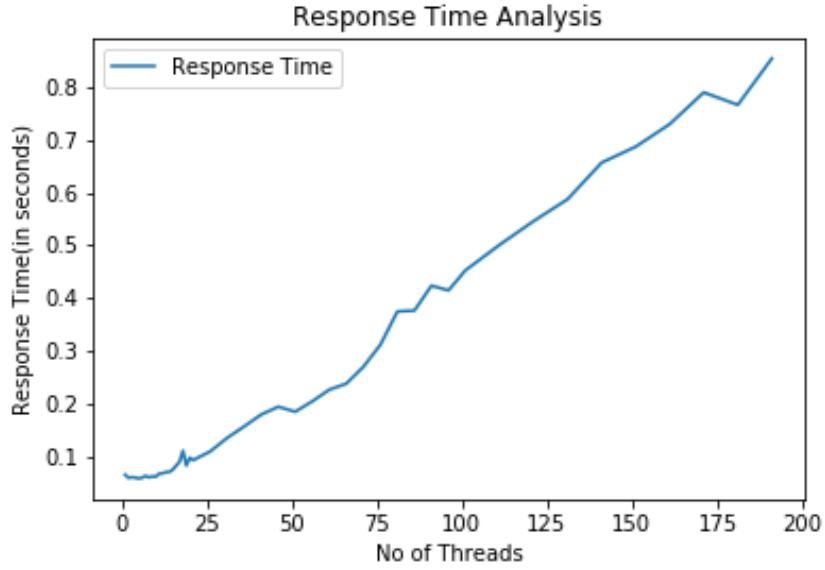

Figure 5: Response Time Graph for 10KB File Download.

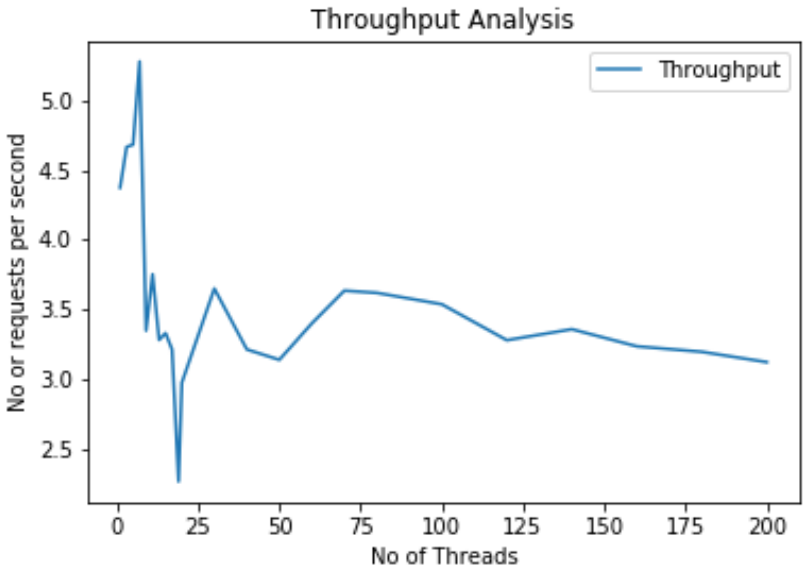

Figure 6: Throughput Graph for 5MB File Download.

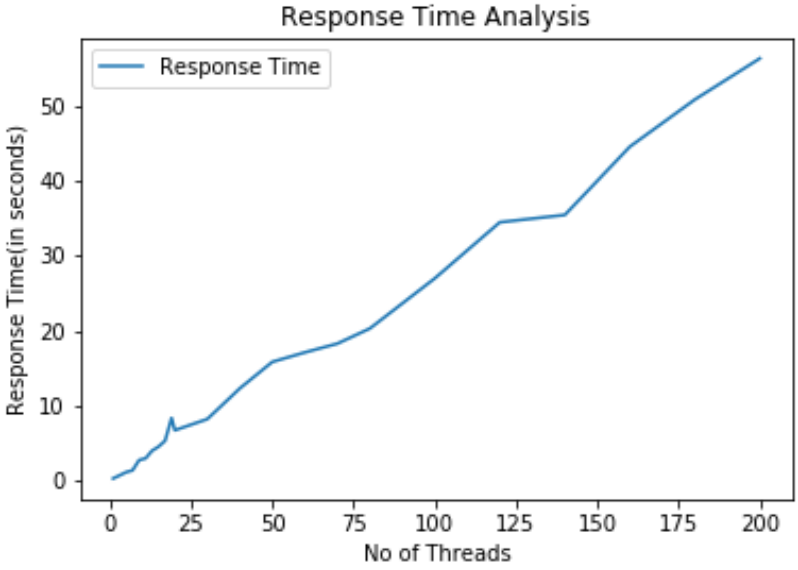

Figure 7: Response Time Graph for 5MB File Download.

As it can be seen from Figure 4 the throughput of our system is increasing till the load generator hits almost 75 no. of users.After that the throughput becomes saturated.We have identified our server's CPU is becoming the bottleneck.After the server hits more than 70 threads the utilization of the CPU reaches 100%. After that the throughput becomes saturated as the CPU is no longer being able to handle more no. of requests simultaneously.

2. Average Response time of System for scenario 1:

   As it can be seen from Figure 5 that the response time is increasing as the number of users are increasing.When the server utilization is full,the system becomes saturated.As the server is saturated and adding more users just increases the response time, while does nothing to the throughput.

3. Average Throughput of System for scenario 2:

   As it can be seen from Figure 6 the throughput of the system remains more or less constant with number of users.This is happening because as the file size is large(5 MB) and we are also cleaning the disk buffer cache simultaneously the I/O capacity (disk read and write at front end server) is being the bottleneck.

   Even if we parallelize with multiple threads IO to a single physical disk is intrinsically a serialized operation. Each thread would have to block, waiting for its chance to get access to the disk. In this case, multiple threads are same single thread and leads to contention problems.

4. Average Response time of System for scenario 2:

   As it can be seen from Figure 7 that the response time is increasing as the number of users are increasing.When the server utilization is full(as the CPU and I/O capacity both becomes bottleneck) ,the system becomes saturated.As the server is completely saturated and adding more users just increases the response time, while does nothing to the throughput.

6 Phase 3 design alternatives and performance analysis

6.1 Event Driven Non Blocking IO

As our project is a file server system, it is mainly a I/O driven application.For I/O we have mainly two options:

1. Blocking I/O
2. non-Blocking I/O

As opposed to Phase 2 we have tried Non-blocking I/O for phase 3.For making I/O non blocking we have used Event-driven I/O mechanism. In case of an Event-driven non-blocking I/O a read or write request always returns instantly

(or near enough). When one try to write, for example, to a socket, the system either immediately accepts what it can into a buffer, or returns something like an EAGAIN or EWOULDBLOCK error letting us know it can't take more data right now.

This is unlike a normal blocking socket where a big write request could block for quite a while as the OS tries to send data over the network to the client or a read request could block if data is not ready in the socket.

In comparison with multi-threaded I/O,it has much reduced overhead in the form of memory, context switching, and takes maximum advantage of what operating systems do best handle I/O quickly.

6.2 Mechanism for Event Driven IO

To implement the Event-driven I/O mechanism we have used - Epoll.The reason why we have chosen Epoll over Select is as follows:

Suppose a typical server might be dealing with, say, 200 connections. It will service every connection that needs to have data written or read and then it will need to wait until there's more work to do. While it's waiting, it needs to be interrupted if data is received on any of those 200 connections.

With select, the kernel has to add the process to 200 wait lists, one for each connection. To do this, it needs a "thunk" to attach the process to the wait list. When the process finally does wake up, it needs to be removed from all 200 wait lists and all those thunks need to be freed.

By contrast, with epoll, the epoll socket itself has a wait list. The process needs to be put on only that one wait list using only one thunk. When the process wakes up, it needs to be removed from only one wait list and only one thunk needs to be freed.

So in contrast with Select which needs linear time to check when an event occurs, Epoll takes almost constant time.

6.3 Difficulties with Epoll

The main problem implementing Epoll(In that case any event-driven architecture) is maintaining states of the server process. In a multi-threaded architecture model the server creates a new thread for every accepted client.The new thread servers the client request.But when we have implemented Epoll the whole server is a single process.So every client which is accepted by the server we have to maintain different state for every client.

Eg. one client has sent a login request.The server was about to read the request buffer.As it is non-blocking I/O before reading this client request another

event has come from another client.Suppose this client is uploading a file.So the server has to save the state of the previous client and will serve the new client's request.As the server handle more requests it becomes very complex to maintain the state of the clients.

To maintain states we have created different map like data structures using vector in c++ where the key of every map is the unique socket fd of that client.And as the values we have stored which states of the client(suppose login) has benn done and which state will be done next(suppose uploading a file).

We have resized all the buffers to 64 Bytes so that the entire buffer fits into 1 cache line and improve the cache performance.

6.4 Graphs

We plot two types of graph for the upload file request as given below with increasing value of load.

1. Average Throughput of System:

   As it can be seen from Figure 8 the throughput of our system is increasing till the load generator hits almost 50 no. of users.After that the throughput becomes saturated and starts decreasing slowly as the number of user increases.This is because here in case of non blocking I/O the server has a single process only.The server process runs a loop and waits for the new events to come.As the number of client as well as the client requests are increasing the process is getting more number of event notifications to serve.The server is being unable to serve more number of requests because most of the requests involve I/O from hard disk(writing file to disk while after receiving from client). I/O from disk can not be served as quickly in case of event driven I/O too.

2. Average Response time of System:

   As it can be seen from Figure 9 that the response time is increasing as the number of users are increasing.When the server utilization is full,the system becomes saturated(Beacause of Disk I/O).As the server is saturated and adding more users just increases the response time, while does nothing to the throughput.

7 Results

1. As the conclusion for phaese 3 we can say that in our case(File server) Event Drive I/O(Epoll) is not performing better than multi-threaded architecture in terms of throughput.Beacause in a file server system the main part of every request (download or upload) is disk I/O(Reading from and writing file to disk).So event driven I/O can reduce waiting time for a process on a non-blocking socket but it won't make much difference when the

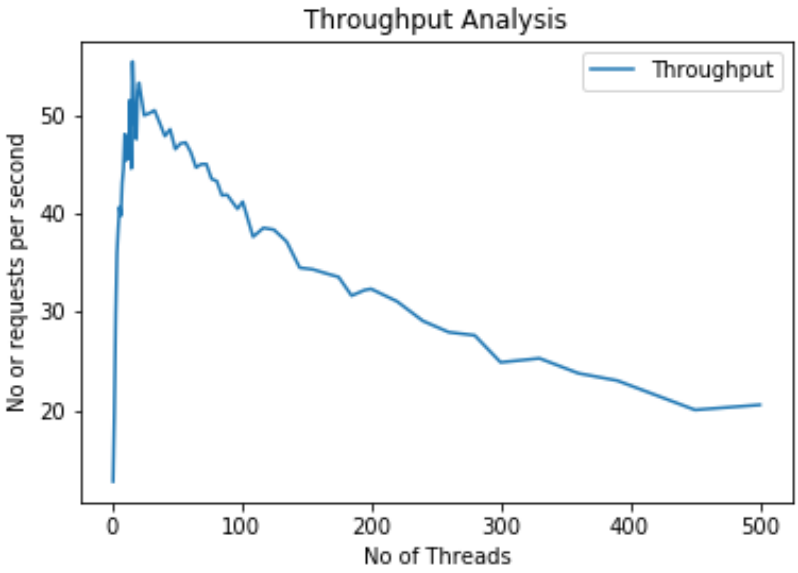

Figure 8: Throughput Graph for 10KB File Upload using Epoll

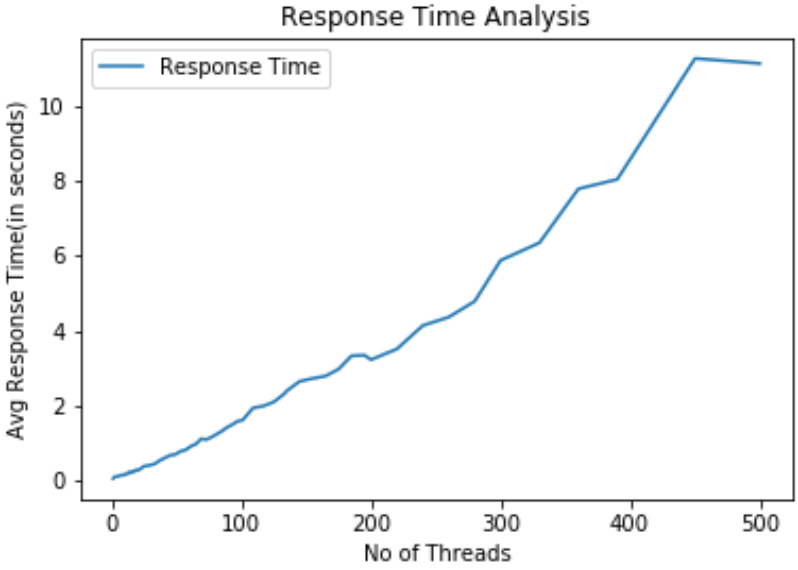

Figure 9: Response Time Graph for 10KB File Upload using Epoll

I/O is from hard disk.Epoll will perform better when the program does not involve much of disk I/O.

2. Epoll program performs better than multi-threaded model in terms memory requirement.Total RAM utilization is less when we run the Epoll enabled server.

3. In terms of scalability also the server is performing better.We are now able to handle much more number of clients when we are using Epoll instead of multi-threaded model.(Beacause of less overhead over all).