COL334 Assignment 2

Dhruv Ahlawat 2021CS10556
Mihir Kaskhedikar 2021CS10551
Payas Khurana 2021CS50948
Tejas Anand 2021CS50595

September 2023

1 Introduction

In this assignment, we attempt to download a text file consisting of 1000 lines from a server using socket programming in python. The server can send only 1 line at a time and has the property that on requesting a line to it, it **gives any line of the text with equal probability**. To speed up the download, we create sockets at multiple clients and implement get and send requests between the clients and the server using threads and locks. The goal of this assignment is to download the file as fast as possible.

2 Strategy

- Our strategy is to make a mesh network amongst ourselves. Though this construction doesn't scale well, as the number of nodes is small (4) in our case it would work fine.
- All of the nodes would individually connect to the vayu server and start downloading the lines on a particular thread. But as soon as vayu stops responding, the node would start requesting files downloaded by other nodes on separate threads.
- We maintain a hash table indexed by line number to keep track of what line numbers we have received.
- Because we have used the mesh topology in our network, the network is fault tolerant and we are able to submit succesfully even if a client disconnects in between. Although, we don't allow the disconnected client to reconnect.

3 Code Explanation

We have initialized the following important global variables -

- `hash_lines`: It is a list of length 1002 and initialized to -1. Whenever we find a new line L having line number l, we assign L to `hash_lines`$[l]$.
- `all_lines`: This is a list of strings where each string is of the form str(l)+'\n'+L+'\n' where l is a line number and L is the corresponding line. At any point this would store all of the different lines that we have obtained from **either the main socket or other clients**.
- `all_my_lines`: This is a list of strings where each string is of the form str(l)+'\n'+L+'\n' where l is a line number and L is the corresponding line. At any point this would store all of the different lines that we have obtained from **only the main socket and not any other client**.
- `servers`: This is a list of sockets present at the IP's of other clients.

and implement the following functions -

- **send_data**: This function takes a socket `socket` and string `data` as input. It encodes `data` (breaks down the string in its byte characters and converts it into an array of bytes) first, sends the size of the encoding, then a character '\r' and then the encoding itself to `socket`. The character '\r' is just used to separate the size of the encoding from the actual encoding and holds no other meaning. (In python it stands for carriage return, but this is not what we intend to do). The size of the encoding is sent so as when `socket` receives this, it knows how many bytes it needs to look for in the incoming `data`.

- **receive_data**: This function takes a socket `socket` as input. It receives the incoming encoded data from `socket` and decodes it. `data` is encoded in the same manner as described in **send_data**: We first receive the number of bytes that `data` contains, and store it in a variable `bytes_to_receive`. Now since the rate at which we receive bytes from `socket` can fluctuate (say due to the strength of network connection), we run a while loop until `bytes_to_receive` turns to 0 and go on appending the decoding of the incoming encoding to `data_array` at every iteration. Finally we concatenate the strings of `data_array` to create `data` and return it.

- **getLinesFromServer**: This function takes a socket `socket` as input (`socket` is a socket situated at one of the other clients). It sends a GET request to this socket using **send_data**, meaning that get lines from this socket. (which lines `socket` sends is described in **clientThread**). Next it receives `data` from `socket` using **receive_data**. Next we split `data` at positions where '\n' appears to get an array of strings `data_array`. This array has the property that data_array[$2i$] stores a line number while data_array[$2i+1$] stores the corresponding line. So we iterate on each line present in `data_array` and check if it is present in `hash_lines` using its line number. If it isn't present, we assign the line to `hash_lines`[corresponding line number]. Also, if the line is not present in `hash_lines`, we create a string concatenation of its line number and itself (and a '\n' in between and in the end) and append it to `all_lines`.

- **clientThread**: This function take a socket `socket` as input (`socket` is situated at one of the other clients). It responds to the GET request made by the other client using **getLinesFromServer** having input as our own socket. Here we have initialized a variable `lastCall` to 0. In every iteration of the while loop, we check if we receive any GET request from `socket`. If we receive it, we iterate on `all_my_lines` starting from `lastCall` till the end and concatenate all of these strings to form `data`. We now update `lastCall` to the size of `all_my_lines` and send `data` to `socket` using the **send_data** function. We have assigned a separate thread for this function, so the while loop is executed for the entire duration of program execution (i.e. until we collect all of the 1000 lines).

- **mainThread**: This function takes the main socket as input (socket created at `vayu.iitd.ac.in`). Here we run a while loop and in every iteration, we send a SENDLINE request to the main socket. If we receive -1 as response, we call the **getLinesfromServer** function **for each of the other client socket**(all of them are stored in the list `servers`). Else if we receive a line as a response, we check if the line is already present in `hash_lines`. If it is not present, we add it to the hash as well as append the concatenation of line number and line to `all_lines` and `all_my_lines`. If size of `all_lines` reaches 1000, we break from the while loop and call the **submitCode** function.

- **myServer**: This function takes a port number `port` and a value `total_connections` as input. Firstly we create a socket at our end on the port `port` and put it in listening mode. Next we accept the incoming connection requests from other clients one-by-one in a while loop (the loop running for `total_connections` times) and create a thread with target as **clientThread** and args as the client and address of the incoming connection. Between accpeting two consecutive connection requests, we introduce a time gap of of **0.1s** using the **time.sleep()** command.

- **submitCode**: This function just sends the collected 1000 lines of the text to the main socket according to the format metioned in the assignment PDF.

- **__name__ == "__main__"**: Here we first create a thread with target `myServer` and args as 12345(`port`) and 3(`total_connections`) (3, if there are 4 clients. In general if there are N clients then we assign the value $N-1$ to `total_connections`) Next we append `total_connections` sockets to the list `servers` and connect these sockets to the servers at the IP's provided in the command line on port 12345. Next we create the main socket and connect it to `vayu.iitd.ac.in` at port 9801. Finally we call the **mainThread** function for this main socket.

4 Theoretical Expectations

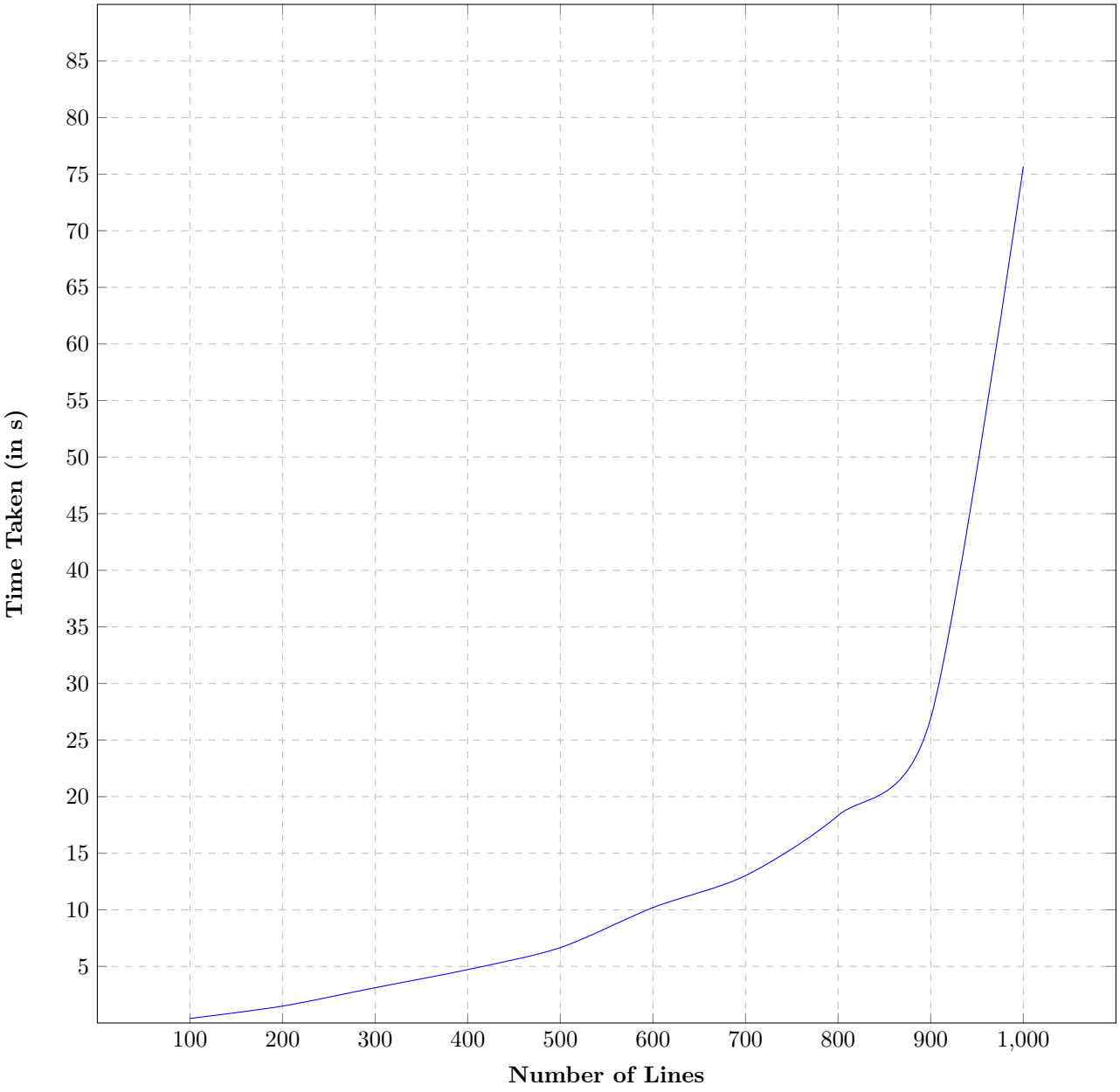
- The theoretical expected requests by each of the servers to get all unique N lines if K servers are downloading the files in parallel is equal to $(\sum_{i=1}^{N} \frac{N}{i}) \cdot \frac{1}{K}$ which is asymptotically equal to $\frac{Nlog(N)}{K}$.

- The expected time required to get total L distinct lines from a total of N lines with K clients where the line transmission limit of server is t lines per second, is $\frac{N}{Kt}(\sum_{i=N-L+1}^{N} \frac{1}{i})$ seconds.

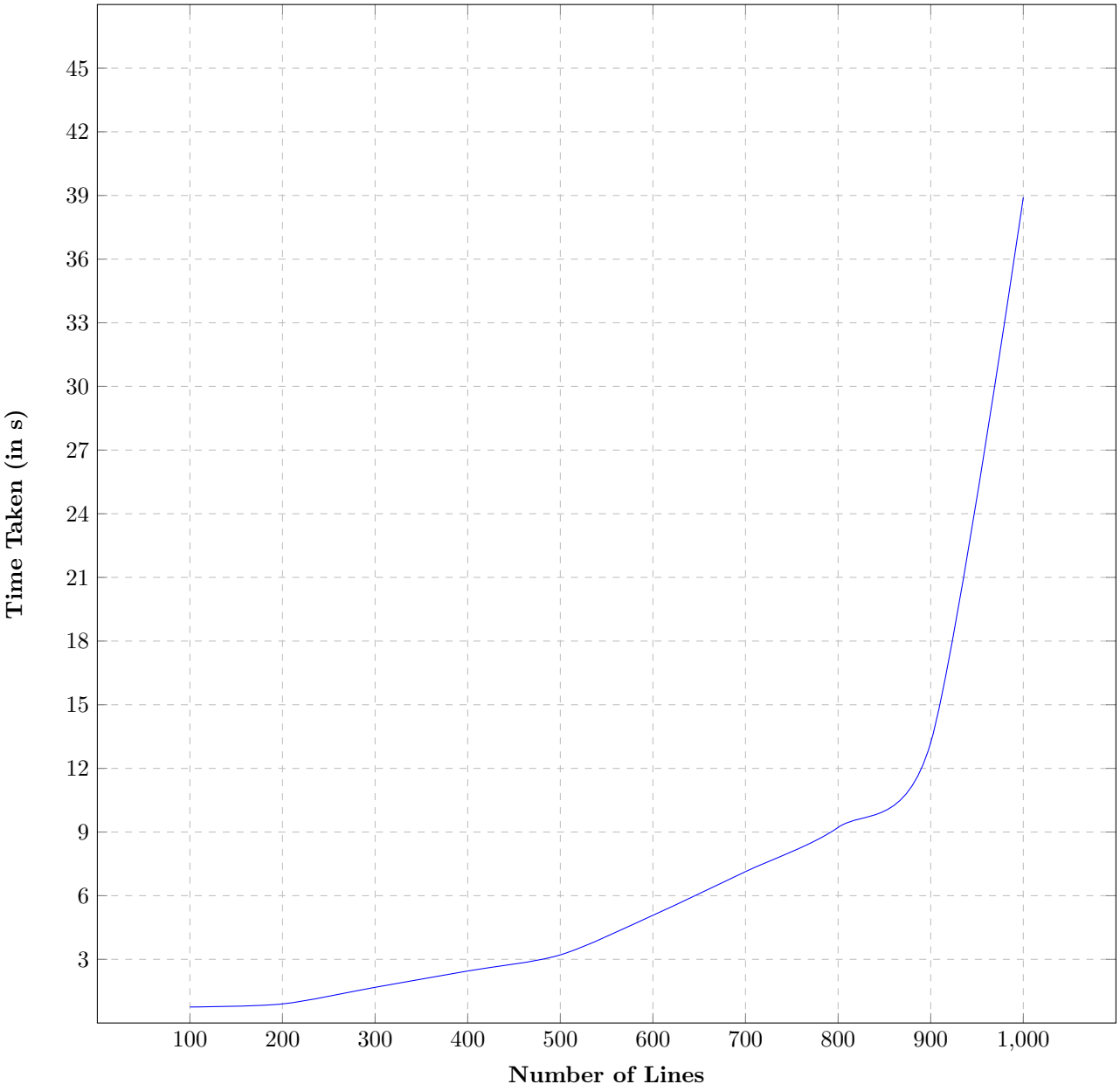
Number of clients	Average time taken for getting all 1000 lines (in s)
1	74
2	37
3	24
4	18

5 Graphs

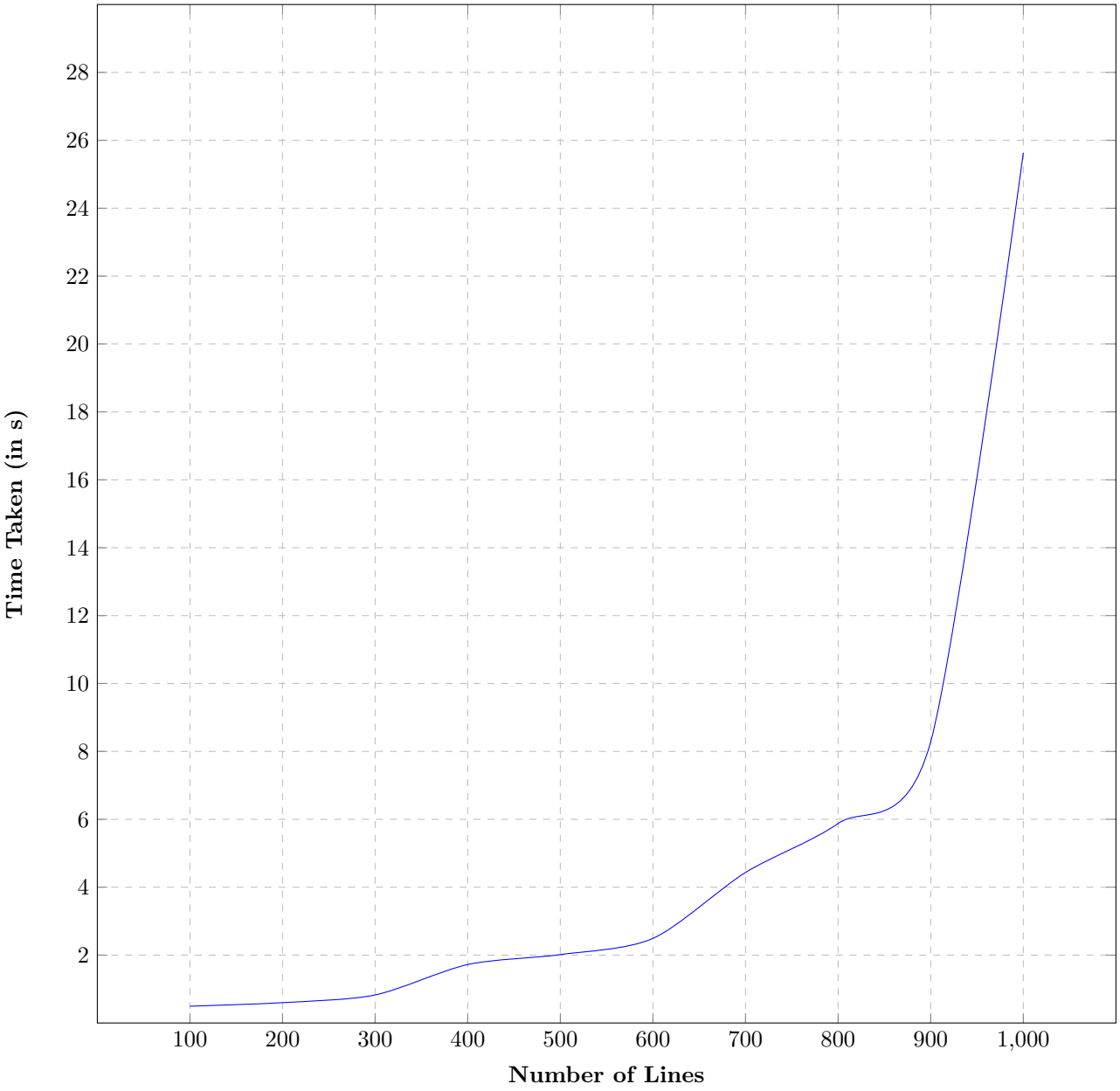
5.1 Number of Clients = 1



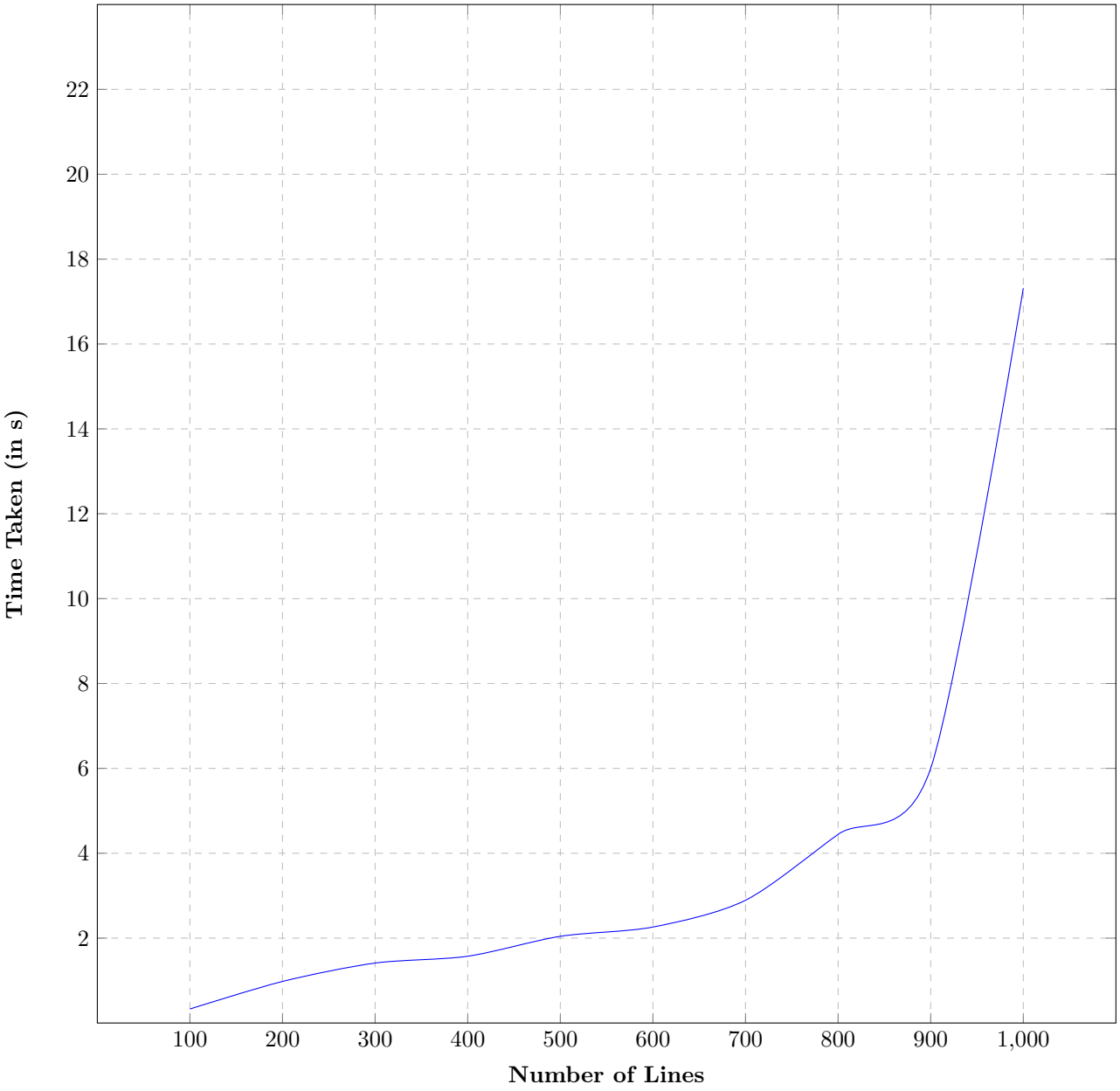
5.2 Number of Clients = 2



5.3 Number of Clients = 3



5.4 Number of Clients = 4



5.5 Comparison