# COL 334
# Computer Networks
# Assignment 3
# Retorrent

Sagar Sharma
2018CS10378

- **To run the assignment open terminal and type**
  **javac download.java**
  **java download [command file name]**

- Some fixed parameters in my assignment:
  **chunk_size = 16kB** except for last chunk which might not fit into 16kB.
  **pipelining = 5**
  **timeout = 5000ms**
  **Note:** My program doesn't stop until the reqQueue becomes empty, that is untill all download size is downloaded. Moreover I have caught and printed error to show the program flow to the grader.
  **Note: All the graphs are made after confirming with hash value given by grader.**

- **Ques 1** is based on single thread download. The whole file is downloaded in 25s. Opened a tcp socket, connected output and input buffers to sockets output and input stream. Then I made an HTTP get request to vayu.iitd.ac.in and the whole file was downloaded. I checked md5 hash of file using my program md5.java and confirmed it was working.

- **Ques 2** is same as ques1 it is just that I incorporated the requests in a loop, each request range is set using a reqQueue (we pop a request and send it in each iteration of loop), in which ranges are stored. Once I recieve a response I parse it from inFromServer stream, and store it in storeQueue, and then the loop repeats. I have been careful and robust about keeping ranges correct even in corner cases.

  Also while setting the reqQueue we need to find the size of file to be downloaded, so I send an HTTP head request to server in beginning, determine the size from header recieved, and then number of chunks is calculated by ceil of size/chunksize.

  At the end I check the md5 hash of file and confirm correct download.

  **Observations on Wireshark**
  On wireshark it is observed that although I have not done pipelining on application level, tcp socket does pipelining of 5 itself. It sends five requests altogether in single packet. The server then sends the response in partial content packets that are reassembled tcp packets to form responsed of a get request. These contents are in order of requests sent.

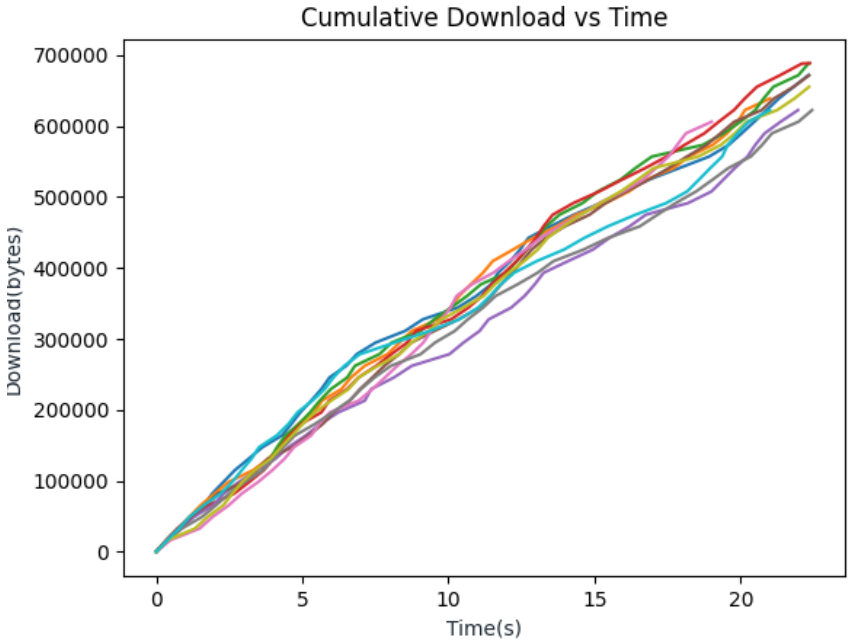
- **Ques 3**
  **Note: from now on, pipelining word is used for pipelining at application level**
  **My Algorithm**
  My program maintains 2 shared resources, reqQueue and storeQueue. In the very beginning i send a Head http request to one of the servers given, so that I can find download size of the file. After parsing HEAD request's response I get the size and then I start making a list in which I store requests to data chunks of 16kB. Then I make multiple threads and run a tcp socket in each of them. These threads pick up 5 requests each from reqQueue form a **GET** request and send it to server. Then I receive the

response, parse it byte by byte and make a pair of (chunk information , chunk data) downloaded and store them in storeQueue. All these resources are synchronised using locks. I keep a save queue for each thread so that if connection breaks, the chunks who's reading was interrupted can be downloaded again, thus the downloading continues from same point after connection reactivates. Later on I sort the storeQueue based on chunk information and get its hash to check if I downloaded correctly.
Other details include; I use try catch block of java in a loop to keep creating sockets and catching exceptions until the file is downloaded completely.
**One more finding was that each thread can't send more than 100 requests, so I restarted the socket in case 100 requests were sent by the thread.**

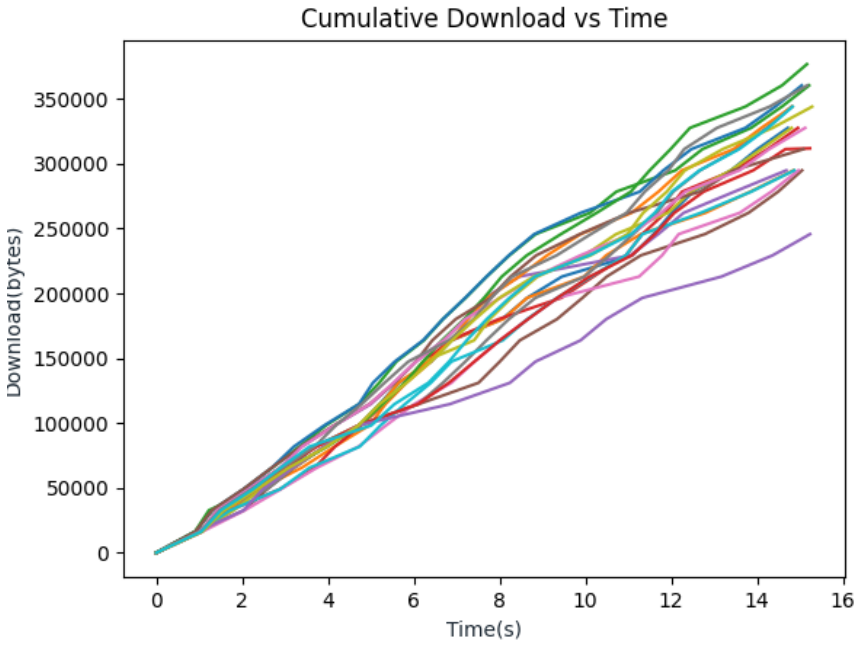
- **3a** The program works for 50-100 threads, though more the threads more the thread starvation as will be discussed later on, and thus doesn't make much difference on increasing threads.

- **3b** As mentioned I created a synchronised shared resource the reqQueue from which each threads picks up a request, makes an HTTP1.1 request out of it and send it to server. There is pipelining of 5 too at application level, that is each thread when given chance to pick will pick 5 requests to send together. After we have parsed and stored the responses to requests sent, we again repeat until the reqQueue becomes empty.

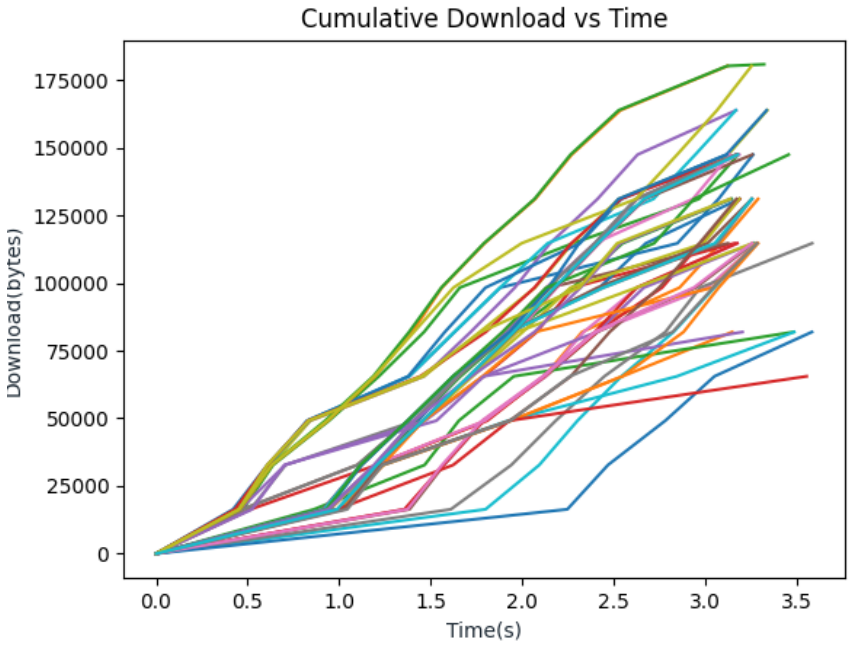
- **3c** Yes the download time keeps decreasing. I experimented with 10,20,50,100 threads. In 50-100 threads, each thread downloads very few chunks. I expected problem of thread overhead, but never the less the download time decreases. There is one curious finding though. As no of threads increase, thread starvation increase. Some threads are stalled more and some less, this ends up with few threads ending up downloading larger fraction of file while rest keep on waiting and remain dormant.
Below are the finding. Downloading from only vayu.iitd.ac.in using different no of threads.



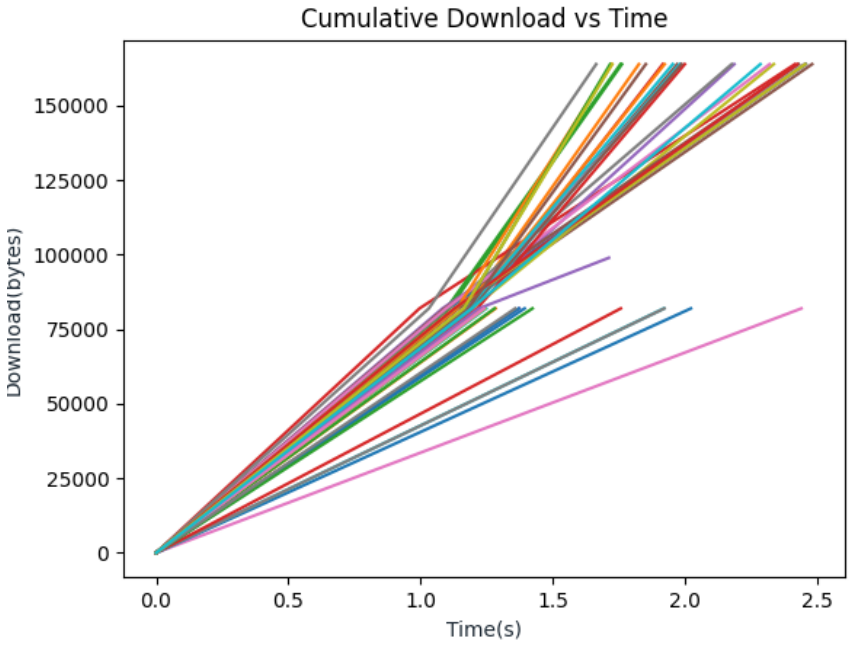

**threads = 10, iitd, pipeline =1**




**threads = 20, iitd, pipeline=1**
This can be observed in the 20 thread example clearly, where some threads download less than others.

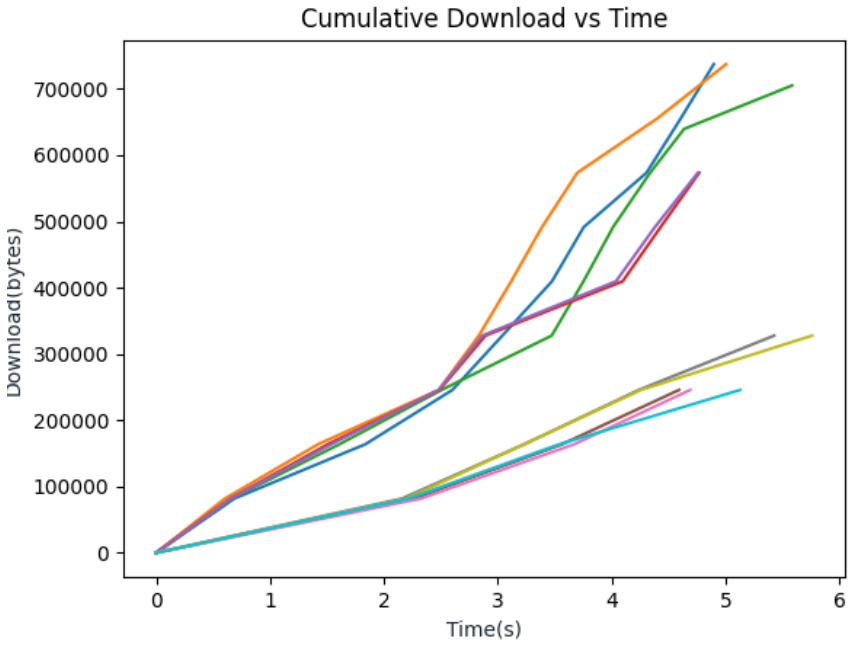



**threads = 50, iitd, pipeline=1**
In case of 50 threads, one can observe the spread of threads, and how threads above have more breakpoints than threads below This clearly shows that above threads were able to pick up more chunks while lower ones starved. This is lot faster but we can claim that increasing number of threads after a certain point becomes unnecessary as thread starvation takes place. Apart from that each thread will also request less number of chunks which has effect on resilience. One can say what if connection keeps on breaking up, the re-download part is lot more in this case.



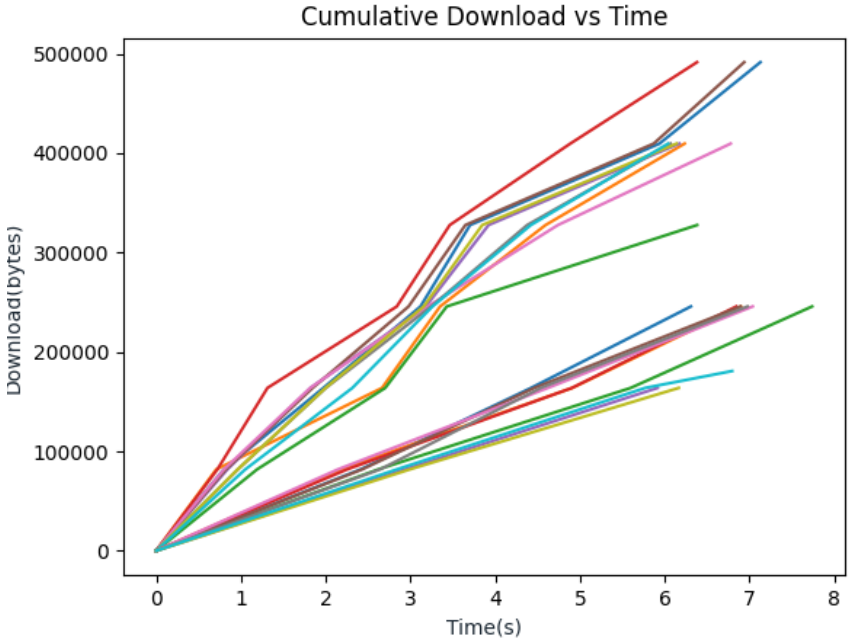

**threads = 50, iitd, pipeline=5**
Apart from all this, pipelining is also there which in turn increases the starvation of threads many folds, as can be seen by the above curve.

- **3d** I experimented with downloading files from norvig + iitd server and found that download time actually increases. Norvig server is not in regional ISP, therefore **rtt** time is high this results in bottle neck bandwidth lower for norvig than iit. iit server is faster as it is in my local ISP. If I would have pre-assigned chunks equally then norvig would have become bottleneck for download but as I used shared resource, it was taken care. The awesome finding related to this was that the program itself takes care of this. It was found that more data was downloaded from iit server than norvig, due to slower speed of norvig, its threads are not able to access the reqQueue as many times the faster dwonloading iit threads are able to.
  Below are my findings



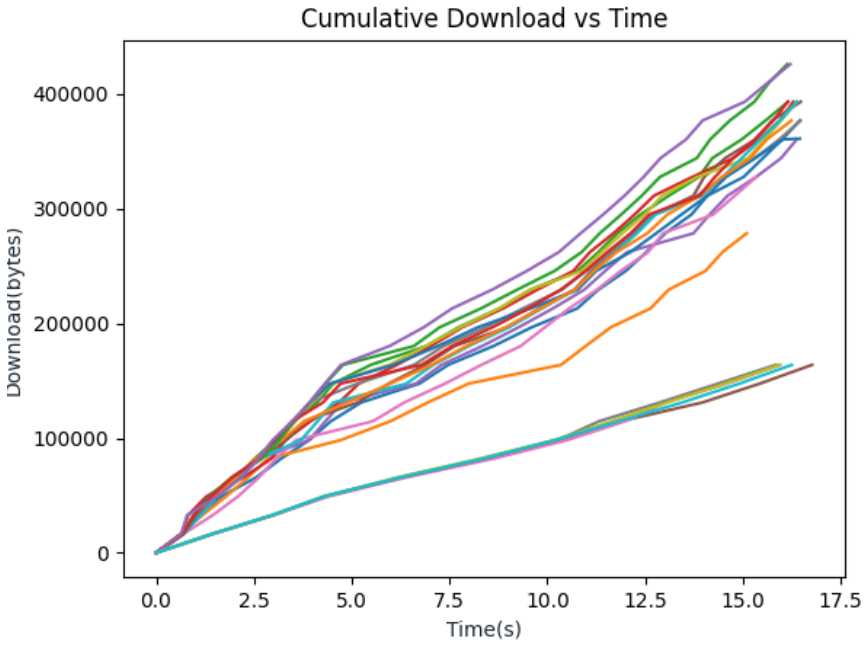

**iitd=5 norvig=5 pipeline=5**

The lower threads are for norvig and the upper ones are for iit. As can be observed the program eventually ends up downloading more from faster server as it doesn't have to wait longer for response and can send more get requests quickly.



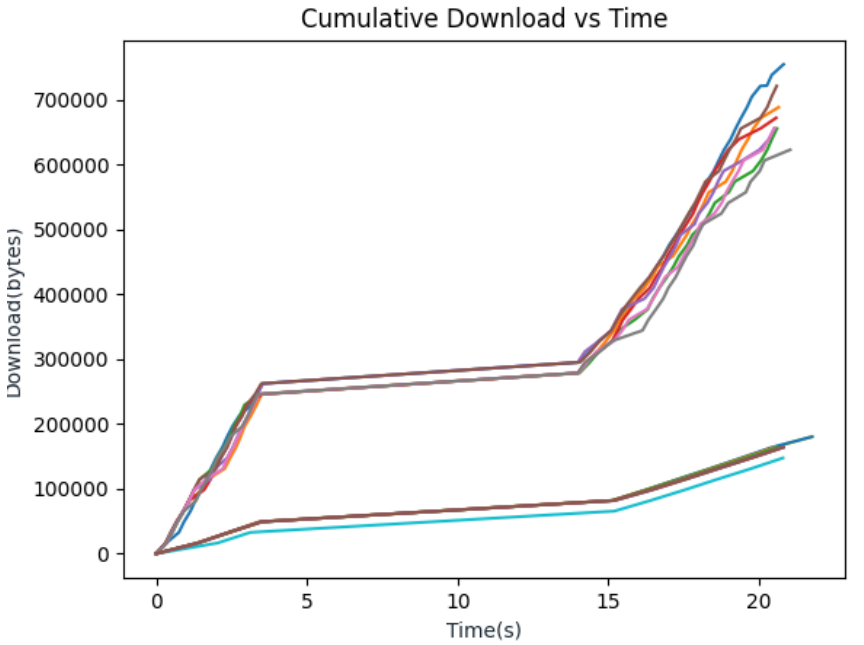

**iitd=10 norvig=10 pipeline=5**

In this one I increases number of threads, same thing is observed, Novig downloads less while iit downloads more. More pipelining means less break points in lower threads, thus Norvig is not able to pick up more chunks than iit from the reqQueue.




**iitd=15 norvig=5 pipeline=1**
Again lower ones are for norvig, upper ones for iit. This one shows that reducing pipeline increases download time.

- **3e** As mentioned I made the program resilient by using try catch block inside while loop. I printed the errors to observe the flow of program. So if an error is caught we simply re run the socket. This keeps on happening until the file is completely downloaded.




**iitd=8 norvig=8 pipeline=1**

This graph shows how the downloading becomes constant, and then again picks up as soon as connection is established.

**Note: All the graphs are made after confirming with hash value given by grader.**