Assignment 2: Mimicking distributed file transfer

COL334/672

In this assignment, we will try to mimic a scenario of multiple nodes cooperating with one another to download a large file.

You will organize yourselves into groups of up to four students, or fewer if you can manage up to four devices amongst yourselves. A server has been setup to which each client can connect to download parts of the file. Each group will have to write a client program and run it on multiple devices (up to four devices) which can collaborate amongst themselves to exchange distinct parts of the file each of them have received and reassemble the entire file.

Some complexities to make this interesting:

- The server is rate limited, i.e. it does not respond to more than a pre-specified maximum rate of requests per unit time.
- The server also allows only one connection per IP address.
- The server is state-less – it responds with a random part of the file and may send the same parts again to a client. The clients will thus be able to reassemble the file quicker when working in cooperation, than if each of them was to download the entire file from the server by itself.

Your job will be to implement the client program and reassemble the file in the shortest possible time. The clients running on each device will connect with the server, download parts of the file, share these parts with one another, reassemble the file, and submit the reassembled file. You will be tracked on how soon you were able to reassemble the file.

To keep it simple, we will use a text file and the server will respond to a client by sending one line at a time in response to requests to fetch a line each time.

Connecting to the server and requesting lines

A test server is running on vayu.iitd.ac.in on port 9801 and accepts TCP connections. You can start experimenting by opening a TCP socket and write the following command into the connection:

SENDLINE\n

In response, the server sends two lines, first the line number and then the line itself. For example:

129\n

This is the 129th line\n

If the rate limit is exceeded, the server sends -1 followed by a blank line:

-1\n

\n

You can also experiment using ncat, e.g.:

ncat vayu.iitd.ac.in 9801

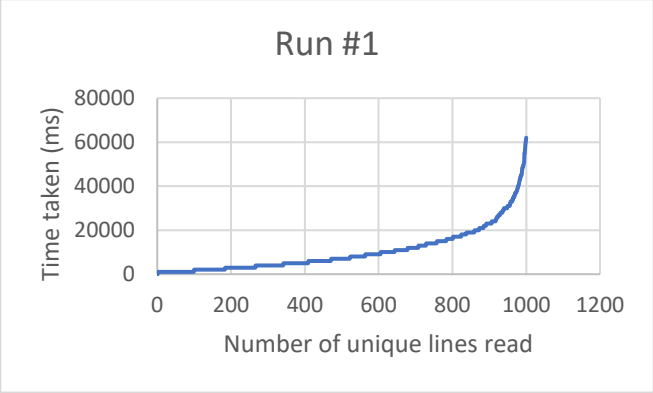
SENDLINE

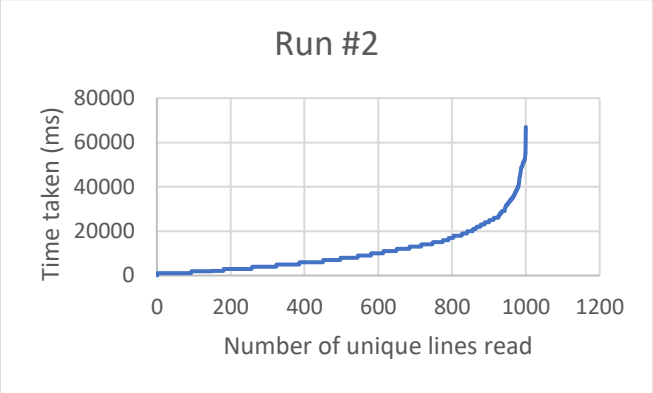
And the server will send a line in response.

The need for multiple clients

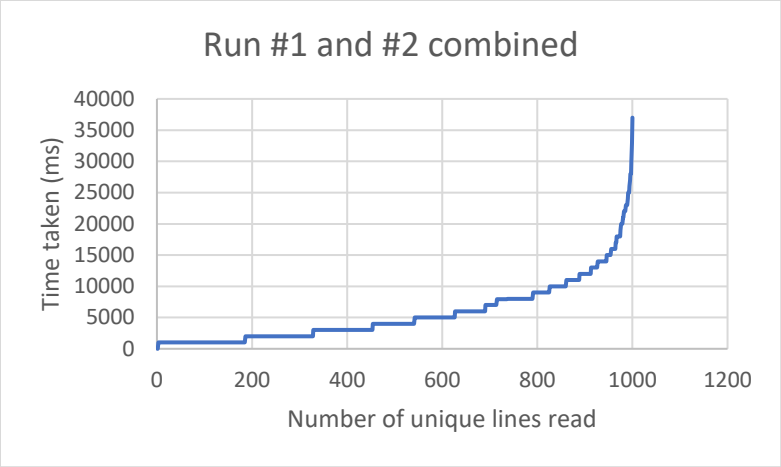
Shown below are outputs from two runs of a simple client program that keeps sending SENDLINE requests and logs the time when a **unique** line is received. The file is 1000 lines long and the server is rate limited to respond with up to 100 lines per second. As you would expect, the rate of receiving unique lines is high in the beginning because the SENDLINE requests are likely to result in the server sending a new line quite often. The rate reduces over time, however, because it becomes less and less likely for the server to send a line that hasn't already been sent earlier.

Here, the client and server were both running on the same host. Receiving the first 600 unique lines took about 10 seconds, the first 800 unique lines took about 18 seconds, but receiving all the 1000 unique lines took more than 60 seconds.

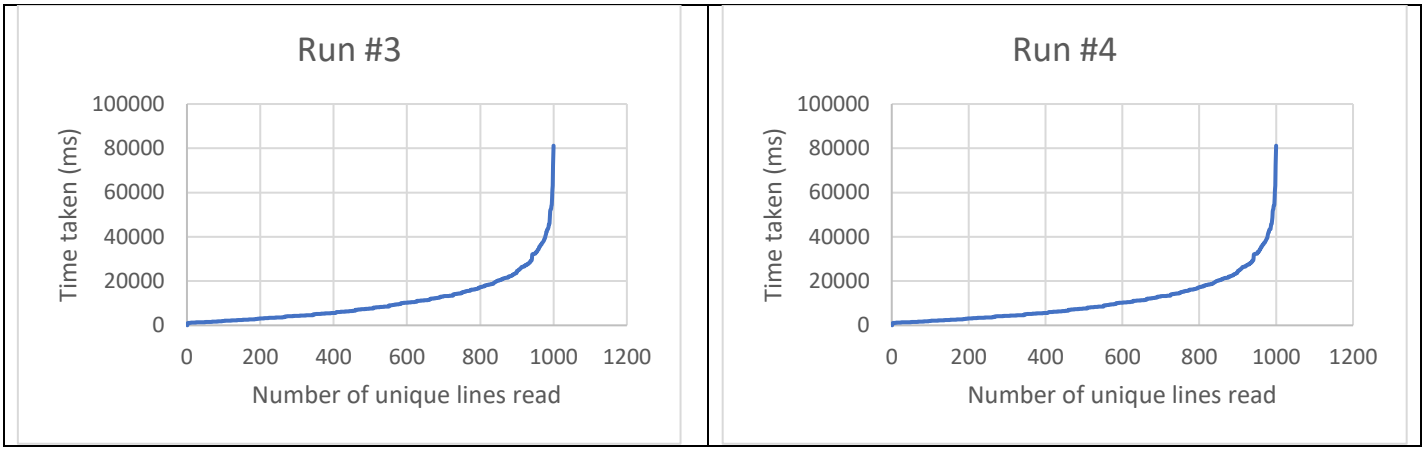


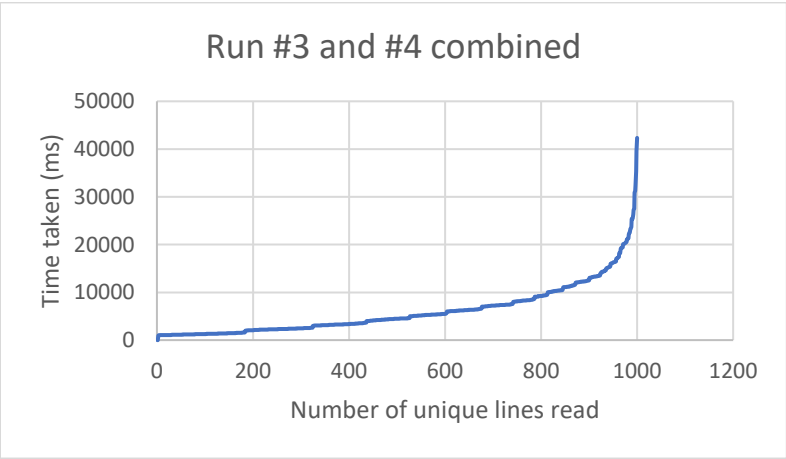


Next, shown is an output from two clients running in parallel and which share their respective received lines with one another. As expected, the chance of getting unique lines at any time is much higher now, and the clients when working in cooperation were able to download the entire file in about 35 seconds.



Similarly, shown below are two runs independently, and then in cooperation, over a WiFi network where the client and server were running on two different devices but connected to the same access point.





Notion of a session

The server allows only one connection per IP address at a time, but it preserves the session state across connections from the same IP address. The server tracks the total number of lines it has sent to the same IP address across multiple connections and the start time of the very first connection from the IP address. This session state is preserved to allow clients to reconnect after a disconnection.

Submitting your entry

Once a client has reassembled the entire file, you can submit it to the server in this manner. On the same TCP socket, you can send a SUBMIT command, followed by your entryID and team name, then the number of lines you are submitting, and then for each of these lines, the line number and the line itself. The server will compare whether the line number and line match the original file and send back an evaluation of the number of lines that correctly matched the original file and the number of lines that did not. All clients in a group should independently submit their entries.

SUBMIT\n

aseth@col334-672\n

5\n

1\n

This is the first line\n

2\n

This is the second line\n

…

If the server was not able to process the input, it may respond with an error:

SUBMIT FAILED: [entryID@teamname] – LINESTOREAD PARSING ERROR\n

Or:

SUBMIT FAILED: [entryID@teamname] – LINE NUMBER PARSING ERROR\n

Or:

SUBMIT FAILED: [entryID@teamname] – READING ERROR\n

A successful submission will result in the following response:

SUBMIT SUCCESS: [entryID@teamname] – [IP address] – [numCorrect] – [numIncorrect] – [numLinesRead] – [numLinesSentSession] – [numLinesSentConnection] – [MAXLINES] – [sessionStartTime] – [connectionStartTime] – [submittedTime]\n

where *numCorrect* is the number of submitted lines in this request that correctly matched the original file, *numIncorrect* is the number of submitted lines that did not match the original file, *numLinesRead* is

the total number of lines submitted and read by the server, *numLinesSentSession* is the total number of lines sent by the server to the IP address across the entire session, *numLinesSentConnection* is the total number of lines sent by the server to the IP address during this current connection, *MAXLINES* is the maximum number of lines (configured to 1000 in this current deployment on vayu.iitd.ac.in), and *sessionStartTime*, *connectionStartTime*, *submittedTime* are the number of milliseconds (long int) counted since January 1, 1970 UTC (coordinated universal time) from the time your client started the session by opening the **first** TCP connection to the server, the time when this connection was opened, and the time you started your submission, respectively. The time you took to reassemble the entire file will be counted as the difference between the *sessionStartTime* and *submittedTime*.

Note that you can submit an entry even before you have downloaded the entire file. The most recent SUBMIT will be considered for evaluation purposes. Each client must make at least one submission.

Implementing such a client-client and client-server network

This is nothing but a straightforward P2P scenario. Below are some hints to get you started.

- Each client program can take commands from you via STDIN. For example, you can write a command
  CONNECTSERVER [IP address of server]
  for the client to connect with the server and start sending SENDLINE commands to it.
- Similarly, you can write commands for the client to connect with other clients
  CONNECTCLIENT [IP address of client]
  and keep sending to them unique lines received from the server, and to read the lines they have received from the server.
- You can implement your own logic for this P2P exchange. For example, you could designate one client as a master which coordinates with all the other clients, or you could connect the clients in a mesh for direct P2P exchanges.
- The regular manner with which each client will operate can therefore be as follows:
  - Open a TCP socket to vayu.iitd.ac.in on port 9801.
  - Open TCP sockets to other clients in your group.
  - Start sending SENDLINE requests to the server and read two response lines each time, containing the line number and the line itself. Record the lines thus received.
  - Share these lines with other clients in your group and receive the lines read by them.
  - Keep track if you have reassembled the entire file.
  - Submit your entry to the server on the same TCP socket.

Forming groups

To the best extent possible, the original groups-of-two formed for the first assignment should be continued for this assignment too, and two of these groups can come together to form a group of four for this assignment. Register your group on this Google doc:
https://docs.google.com/spreadsheets/d/1GzLKpaxDs-ykCrQ2Aofz_M4xCHu0Gsp7MsvFCQwwgYA/edit?usp=sharing

Evaluation

There will be three components to the evaluation:

- Submit your code and an assignment report on Moodle. In the report, clearly explain the distributed algorithm you have implemented for the P2P exchange. Submit graphs of the sort shown here from test runs, with one client downloading the file by itself, with two clients working in cooperation, then three clients, and finally four clients. Does the download time reduce linearly as you add more and more clients?

  Also report various exception scenarios you handle, for example, if a client node in your P2P network suddenly dies and wants to reconnect, or if a client-server connection breaks and the client reconnects, etc.

- Participate in a viva with your TA mentors. Students should attend the viva with their original TA mentors, and make sure that they have at least two devices between themselves to demo the distributed setup.

- Participate in a tournament! We will run a tournament during one of the classes and see who emerges as the winner! The structure of the tournament will be communicated in due course after we have instrumented the servers to understand how many parallel connections a single server is able to handle and how many servers are we able to put up.