Lab 4: Rate-limited FTP

Due date: Thu May 1, 2025 (11:59 PM)

TA in charge: Rohit Pai

Updates:

- 04/24/25 12:48 PM - Clarified the directory argument for the server. Feel free to use the "filepath" package in Go to deal with paths and avoid OS-specific differences.
- 27 April 3:35 PM: Fixed the example commands to incorporate correct use of port/host flags

In this exercise, you're going to implement a very simple, and VERY insecure file transfer service. It supports two commands, **`RETR`** (which retrieves a file from the server and downloads it to the client), and **`STOR`** (which uploads a file from the client to the server). When the client uploads a file to the server, it should store that file in a single directory (provided as a configuration parameter on the command line). Likewise, when a client requests a file from the server, the server will find that file in that same directory. Thus, there are no subdirectories–just files. You do not need to support listing files, renaming files, or other operations that a real networked file program would have to support. Just upload and download files.

In addition to sending/receiving files, your server will support *rate-limiting* the retrieval of files. Clients can request a rate limit of the file in units of "bits per second" and the server will send data to the client at as close to the desired rate as possible.

Starter Code

- To begin the project, please accept the GitHub invitation and clone your repository. Don't forget to commit and push your code when you're done. Finally link your github repo with GradeScope.
- Github invitation: https://classroom.github.com/a/beBFIliL

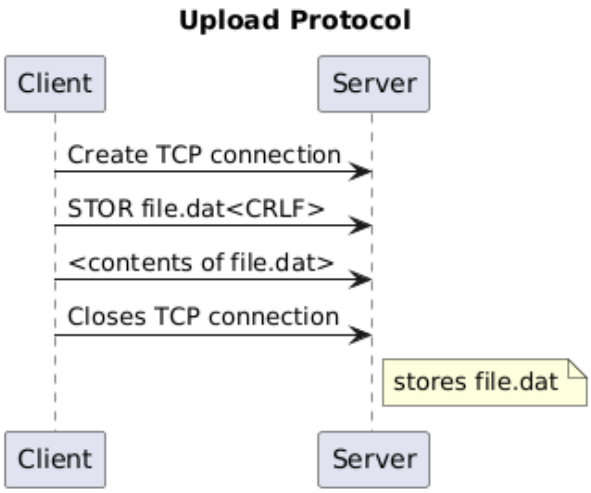
Description

Your server will open a listening socket and wait for an incoming connection. When a client arrives, it will issue a single command (and provide the file data in case of **`STOR`**) and then close the connection. Likewise, when the server is finished handling an incoming command, it closes the socket and waits for the next client. Your server should be concurrent/parallel, meaning that multiple clients should be able to connect (and download/upload) files in parallel. Each client can specify its own rate limit, which the server should honor.

You can assume that any files to download already exist on the server, that the permissions are all correct, and that the client doesn't ask for non-existent files.

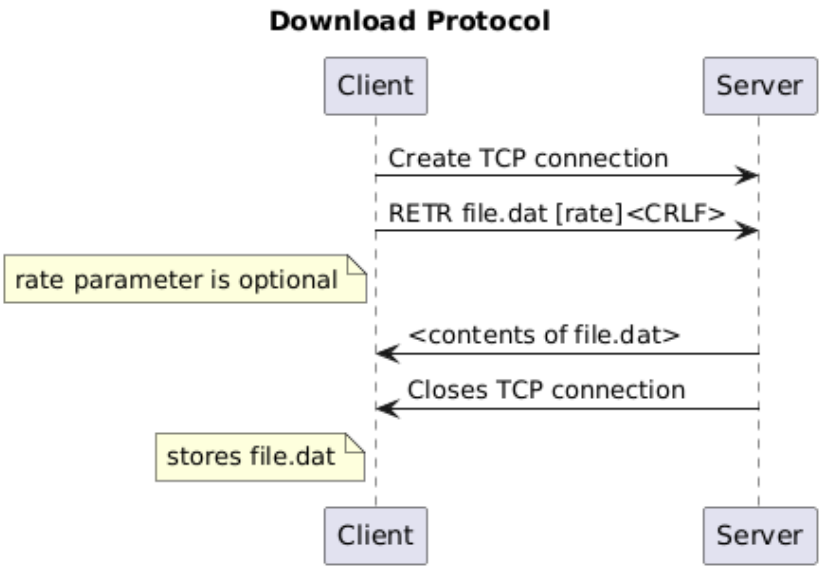
Protocol: Upload

To upload a file, the client connects to the server, issues a **`STOR`** command and provides the filename. The <**`CRLF`**> delimiter signifies the end of the **`STOR`** command. Then, the client provides the contents of the file that need to be stored on the server and closes the connection. After receiving the file contents, the server stores the data as the specified filename on its local storage.



Protocol: Download

To download a file, the client connects to the server, issues a **`RETR`** command and provides the filename. The <**`CRLF`**> delimiter signifies the end of the **`RETR`** command.



Client and server programs

The starter code includes a server and the client starting point. In general, they are invoked as follows:

The client

Usage: ftpclient <STOR|RETR> <filename> [rate]

- You can specify --host and --port options

* “-host string” The ip address or hostname of the server
* “-port p”: The port the server is listening on
* [RETR|STOR]: indicates whether you want to upload or download a file
* filename: if uploading, the name of the file you want to upload (you can assume that we will pass in an absolute path to the filename--do not use a relative path or assume the file is in the current directory). If downloading, the name of the file you want to download. Note that the server doesn't support subdirectories, so if you download a file you just give it the name of the file (no path). When downloading, the client should store the file in the current working directory.

You do not have to handle error conditions such as trying to download a file that doesn't exist on the server, or trying to upload a file that doesn't exist on the client.

Here are some examples:

```
```

$ ./client --host 127.0.0.1 --port 8080 STOR /tmp/myfile.txt
$ ./client --host 127.0.0.1 --port 8080 STOR /home/users/turing/sample-file.dat
$ ./client --host 127.0.0.1 --port 8080 RETR myfile.txt
$ ./client --host 127.0.0.1 --port 8080 RETR sample-file.dat 50000
```

## The server

Usage: ./ftpserver <directory>

* -host: the host address to listen on
* -port: the port to listen on
* <directory>: Defines the path to the directory where the server is supposed to store and retrieve the files from (you can assume that we will pass in an absolute path to the directory--do not use a relative path or assume it would just be the current directory)

## Rate limit

The client can optionally specify a rate limit, in bits per second, when retrieving files. You can implement this rate limit a few ways. However, one way is to have your server send chunks of the file over the socket to the client, while waiting an appropriate amount between chunks. The size of your file chunks is up to you in this case, but to ensure you don't invoke too many system calls, you should probably send at least a kilobyte per chunk.

As the requested rate gets higher and higher, it becomes increasingly difficult to wait the precise amount of time between sending each chunk of data. At very high speeds, implementing network rate limiters in software requires a different approach. We will only request rate limits under 250000 bits per second (250 kilobits per second).

To evaluate the accuracy of your rate limit, you can add logging and timing code to your client. You can also run the client inside of the Linux `time` command, to provide a second datapoint.

## File Check

To check whether your client/server are working correctly, you can use the `diff` command in linux. Simply upload a file, then download that file, and compare the two. As an example:

```
$ cd /tmp
tmp $ client --host localhost --port 8080 STOR
/home/users/aturing/sample-file.dat
tmp $ client --host localhost --port 8080 RETR sample-file.dat
tmp $ diff /tmp/sample-file.dat /home/users/aturing/sample-file.dat
```
```

Type `man diff` for more information.

Evaluation

We will run your client connecting to our server, or our client connecting to your server, or your client connecting to your server (testing will vary). We'll run a set of different tests where we upload some set of files, download from that set of files, and compare the results. The files we'll test against will include a mixture of text and binary data, and a mixture of sizes (small, medium, and large). You should support files up to 10s of megabytes in size. In addition to correctness tests, we will also download files using different rate limits and ensure that the observed download time is within a specific range of the expected download time given the provided rate limit.