CMPE 156/L, Winter 2019

Programming Assignment 3
Due: _____

The problem in this assignment is to develop the client and server sides of an application that enables a client to download "chunks" of a file from multiple servers (for example, ftp mirrors) distributed over the Internet, and assemble the chunks to form the complete file. When the server is the bottleneck, this can speed up the download of a large file over normal ftp.

To test this application, you will have a single client process on your machine opening connections to multiple servers. The design should allow the server processes to run on any machine on the Internet, although for testing you can run several server processes on the same machine. The client will obtain the contact information for the servers from a text file server-info.text that lists the IPv4 addresses of the server hosts and the corresponding listening port numbers.

128.32.16.1 4321
192.71.62.5 5432
127.0.0.1 5555

The IP address 127.0.0.1 represents "localhost," indicating that the server is on the local machine. You should start each copy of the server using the command line.

./myserver <port-number>

where the argument is its listening port number. The client process is started with the command

./myclient <server-info.txt> <num-connections> <filename>

The client client should first contact one of the servers listed in server-info.text and obtain the size of the file, named <filename> to be transferred. It should then connect the number of servers as specified in the <num-connections> argument and transfer a chunk of the file from each server. Each chunk is specified by a starting offset and a chunk size. The client must calculate these parameters by dividing the size of the file by the number of connections. The client must then initiate the connections and transfer each chunk concurrently. When all the chunks have been received, it should assemble them into the complete file and save it on disk (so that you can do diff with the original file).

Requirements

- The server does **not** need to be concurrent. If you design it as concurrent, it needs to have the ability to send the same file or different files to multiple clients in parallel.
- You should use threads based on the Pthreads library to implement the client.
- You need to specify the commands, responses, message formats, etc., to be used by the client and server to achieve the desired functionality. At a minimum, the client needs to check if a certain file exists at the server, and get its size. The client should also be able to initiate the transfer of a chunk of the file, identified by a starting offset and chunk size. You should also take care of any error scenarios that can arise in the client-server interactions. You may borrow ideas from the TFTP protocol.
- The client can choose the servers to use from those listed in server-info.text in any order. If the number of servers specified in the <num-connections> is more than the number of servers available, the client should determine the actual number of connections as the smaller of the two.
- If one of the selected servers happens to be unreachable or malfunctions, the client must use an alternate server to transfer the corresponding chunk. If all the servers in server-info.text are already in use, the client must wait for the transfer of one of the chunks to be completed and use this same server to transfer a second chunk.
- All data between the client and a server must be exchanged over a single TCP connection (that is, no separate data and control connections as in ftp). It is acceptable to use the OOB option of TCP.
- The server design must be robust against errors and network failures. The application should not fail unless the servers listed in server-info.text have all failed.
- The server should have a number of files available to serve (at least 6). The files can be limited to text files, and their total sizes can be limited to 1000 Kbytes. These files are to be placed in the same directory where the server program is.

What to submit?

You must submit all the files in a single compressed tar file (with tar.gz extension). The files should include

1. All source code necessary to build and run the client and server.
2. A README file including your name and a list of files in the submission with a brief description of each. If your code does not work completely, explain what works and what doesn't or has not been tested.
3. A Makefile that can be used to build the client and server binaries.
4. Documentation of your design in plain text or pdf. Do not include any Microsoft Word files. The documentation should describe the internal design of your client and server implementations.
5. Including in the design document, a specification of the application-layer protocol, describing the handshakes involved, message formats, error handling, etc.
6. Organize the files into directories (src, bin, doc, etc.)

Grading

Each submission will be tested to make sure it works properly and can deal with errors. Grades are allocated using the following guidelines:

Basic functionality	20%
Basic testing	20%
Dealing with errors	30%
Documentation	15%
Style/Code structure, etc.	15%

Note that 30% of the grade will be based on how well your code deals with errors. Good practices include checking all system calls for errors and avoiding unsafe situations such as a buffer overflow.

Honor Code

The files must be developed independently. All the work submitted must be your own.